Merlin™

by Glen Bredon

INSTRUCTION MANUAL

Copyright © 1982 by Southwestern Data Systems. All rights reserved. This document, or the software supplied with it, may not be reproduced in any form or by any means in whole or in part without prior written consent of the copyright owners. SC282JCL

PRODUCED BY:

SDS™

SOUTHWESTERN DATA SYSTEMS™

10159-I Mission Gorge Road • Santee, California 92071

Telephone: 714/562-3670

CHRGEN 70 is a recent addition to your MERLIN assembler package, and as such is not described in the manual accompanying the diskette. CHRGEN 70 is a 70-column character generator which is designed specifically to allow the use of MERLIN with a 70 by 24 display on the Hi-Res screen. If you have an 80-column display card, CHRGEN 70 will not be much slower than with any hardware display.

TV sets do not provide sufficient resolution for use with CHRGEN 70, thus requiring use of a display monitor for satisfactory results.

To use CHRGEN 70 you must first BRUN it from MERLIN's EXEC mode as a DOS command (after a CATALOG). This will reset the source address to $4001 (above the Hi-Res screen which must be used by CHRGEN 70). This, of course, will delete any source file in memory at the time. Once it has been BRUN, you can invoke it at any time by typing "USER" from the editor.

To exit CHRGEN 70, simply type VID 0, VID 16, or PR#0 from the editor. CHRGEN 70 is automatically disconnected when you exit the editor to the EXEC mode. Upon return to the editor, you can reconnect it by typing "USER" again. To permanently remove CHRGEN 70 in order to free up the area normally used by long source listings, you will have to BRUN MERLIN again.

To use CHRGEN 70 with the editor's PRTR command, just type PRTR 8 "filename", with CHRGEN 70 installed in the system.

If the USER vector has been written over by some other USER routine, it can be reset to point to CHRGEN 70 either by BRUNing CHRGEN 70 again, or by going to the Monitor (use the MON command) and typing in 900G. The latter assumes, of course, that CHRGEN 70 is still intact at $900.

CHRGEN 70 includes a version of the FORMATTER program. To implement FORMATTER when CHRGEN 70 is connected, just type Control-T from the editor's command mode. NOTE: This command may not be accepted unless something has been listed previously.

CHRGEN 70 also includes some keyboard macros. Typing the ESCAPE key followed by certain other keys will produce the keypoard macros. These are presently defined for these keys as:

```
* > - ' f +
: . 2 7 3 ; X Y D H P @ L S - O C A E
```

The macro table lies at the end of the CHRGEN 70 program at $1500 and is modifiable. It must end with a $FF.

CAUTION: When CHRGEN 70 is up, you must not load any binary source file longer than 88 sectors or it will overwrite the DOS buffers and bomb the system. Text files do not present this danger since they are never allowed to go beyond HIMEM.

SOUTHWESTERN DATA SYSTEMS CUSTOMER LICENSING AGREEMENT

This software product is provided to you subject to the terms and conditions of this agreement. If you feel you cannot accept these terms and conditions, you must return this product intact within 10 days of original purchase.

1. LICENSE. Southwestern Data Systems hereby grants you upon receipt of this product, a nonexclusive license to use the enclosed product subject to the terms and restrictions set forth in this agreement.

2. COPYRIGHT. This software product and its documentation, is copyrighted by Southwestern Data Systems and program author.

3. RESTRICTIONS ON USE AND TRANSFER. The original and any back-up copies of this product are intended for your personal use in conjunction with a single computer. You may not copy or otherwise reproduce this product in whole or part except as expressly permitted in this agreement.

D I S C L A I M E R

SOUTHWESTERN DATA SYSTEMS AND THE PROGRAM AUTHOR SHALL HAVE NO LIABILITY OR RESPONSIBILITY TO PURCHASER OR ANY OTHER PERSON OR ENTITY WITH RESPECT TO ANY LIABILITY, LOSS OR DAMAGE CAUSED OR ALLEGED TO BE CAUSED DIRECTLY OR INDIRECTLY BY THIS SOFTWARE, INCLUDING, BUT NOT LIMITED TO ANY INTERRUPTION OF SERVICE, LOSS OF BUSINESS OR ANTICIPATORY PROFITS OR CONSEQUENTIAL DAMAGES RESULTING FROM THE USE OR OPERATION OF THIS SOFTWARE.

CHRGEN 70 also includes some keyboard macros. Typing the ESCAPE key followed by certain other keys will produce the keypoard macros. These are presently defined for these keys as:

```
* > " ' { +
: . 2 7 3 ; X Y D H P @ L S - O C A E
```

The macro table lies at the end of the CHRGEN 70 program at $1500 and is modifiable. It must end with a $FF.

CAUTION: When CHRGEN 70 is up, you must not load any binary source file longer than 88 sectors or it will overwrite the DOS buffers and bomb the system. Text files do not present this danger since they are never allowed to go beyond HIMEM.

LISA V2.5 → MERLIN

```
CW "EPZ"EQU"
CW "ADR"DA" "
CW "DFS"DS "
```

TABS
TEXT

MERLIN TABLE OF CONTENTS

TITLE	SECTION-PAGE
INTRODUCTION	
Assembly Language Whys and Wherefores	1 - 1
Background and Features	1 - 3
Suggested Readings.	1 - 4
System Requirements	1 - 6
Compatibility List.	1 - 6
Beginners Guide To Using MERLIN	1 - 7
Introduction	1 - 7
Input	1 - 8
System and Entry Commands.	1 - 11
Assembly	1 - 14
Saving and Running Programs	1 - 16
Making Back-up Copies of MERLIN	1 - 18
EXECUTIVE MODE	2 - 1
Catalog	2 - 1
Load	2 - 1
Save	2 - 2
Append.	2 - 2
Drive	2 - 2
Editor.	2 - 2
Save Object Code.	2 - 3
Quit	2 - 3
Read	2 - 3
Write	2 - 3
THE EDITOR	3 - 1
Command Mode.	3 - 1
HIMEM	3 - 1
New	3 - 1
PR#	3 - 2
USER	3 - 2
TABS	3 - 2
LENgth	3 - 2
Where	3 - 2
MONitor.	3 - 3
TRuncON.	3 - 3
TRuncOFf	3 - 3

Quit . . . . . 3 - 3
ASM. . . . . 3 - 3
Control-D. . . . . 3 - 4
Delete . . . . . 3 - 4
Replace. . . . . 3 - 4
List . . . . . 3 - 4
Period (.) . . . . . 3 - 5
Print . . . . . 3 - 5
PriNTeR. . . . . 3 - 5
/ . . . . . 3 - 6
Find . . . . . 3 - 6
Change . . . . . 3 - 6
COPY . . . . . 3 - 6
MOVE . . . . . 3 - 7
Edit . . . . . 3 - 7
Hex-Dec Conversion . . . . . 3 - 7
TEXT . . . . . 3 - 7
FIX . . . . . 3 - 8
SYM . . . . . 3 - 8
VIDeo . . . . . 3 - 9
FW . . . . . 3 - 9
CW . . . . . 3 - 9
EW . . . . . 3 - 9
VAL. . . . . 3 - 9

Add/Insert Mode . . . . . 3 - 10
Add . . . . . 3 - 10
Insert . . . . . 3 - 10
Control-L . . . . . 3 - 10

Edit Mode . . . . . 3 - 11
Control-I . . . . . 3 - 11
Control-D . . . . . 3 - 11
Control-F . . . . . 3 - 11
Control-O . . . . . 3 - 12
Control-P . . . . . 3 - 12
Control-@ . . . . . 3 - 12
Control-C . . . . . 3 - 12
Control-X . . . . . 3 - 12
Control-B . . . . . 3 - 12
Control-N . . . . . 3 - 12
Control-R . . . . . 3 - 13
Control-Q . . . . . 3 - 13
RETURN . . . . . 3 - 13

ASSEMBLER . . . . . 4 - 1
Number Format . . . . . 4 - 2
Source Code Format . . . . . 4 - 3
Expression . . . . . 4 - 4
Immediate Data . . . . . 4 - 5
Addressing Modes . . . . . 4 - 5
6502 OpCodes . . . . . 4 - 6
Sweet 16 OpCodes. . . . . 4 - 6
Psuedo Opcodes . . . . . 4 - 6
Directives
EQUals . . . . . 4 - 7
ORiGin . . . . . 4 - 7
OBJect . . . . . 4 - 7
PUT . . . . . 4 - 8
VAR . . . . . 4 - 8
SAV . . . . . 4 - 9
DSK . . . . . 4 - 9
END . . . . . 4 - 9
Formatting
LiST . . . . . 4 - 10
EXPand . . . . . 4 - 10
PAUse . . . . . 4 - 10
PAGe . . . . . 4 - 10
ASTerisks. . . . . 4 - 10
SKiP . . . . . 4 - 11
TR . . . . . 4 - 11
Strings
ASCii . . . . . 4 - 11
Dextral Character Inverted . . . . . 4 - 11
INVerse . . . . . 4 - 11
FLaSh . . . . . 4 - 12
REV . . . . . 4 - 12
Data and Allocation
Define Address . . . . . 4 - 12
Define Double Byte . . . . . 4 - 12
DeFine Byte . . . . . 4 - 13
HEX . . . . . 4 - 13
Define Storage . . . . . 4 - 13
KBD . . . . . 4 - 13
LUP . . . . . 4 - 14
CHK . . . . . 4 - 14
ERR . . . . . 4 - 15
USR . . . . . 4 - 15

Conditionals
DO . . . . . 4 - 17
ELSE . . . . . 4 - 18
FIN . . . . . 4 - 18

Macros
MACro. . . . . 4 - 18
EOM . . . . . 4 - 19
Put MaCro. . . . . 4 - 19
Variables . . . . . 4 - 19

MACROS
Defining a Macro. . . . . 5 - 1
Nested Macros . . . . . 5 - 1
Special Variables . . . . . 5 - 3
Sample Program . . . . . 5 - 5
The Macro Library . . . . . 5 - 6

TECHNICAL INFORMATION
General Information . . . . . 6 - 1
Memory Map . . . . . 6 - 3
Symbol Table. . . . . 6 - 4
Using MERLIN with Shift Key Modification. . . . . 6 - 4
Using MERLIN with 80 Column Boards. . . . . 6 - 4
The Configure ASM Program . . . . . 6 - 5
Error Messages . . . . . 6 - 7

SOURCEROR
Introduction . . . . . 7 - 1
Using Sourceror . . . . . 7 - 1
Commands Used in Disassembly
List . . . . . 7 - 3
Sweet . . . . . 7 - 4
Normal . . . . . 7 - 4
Hex . . . . . 7 - 4
Text . . . . . 7 - 4
Word . . . . . 7 - 5
Housekeeping Commands
Cancel . . . . . 7 - 5
Read . . . . . 7 - 5
Quit . . . . . 7 - 6

Final Processing . . . . . 7 - 6
Dealing With The Finished Source . . . . . 7 - 7
The Memory Full Message . . . . . 7 - 8
The LABELER Program . . . . . 7 - 8

SWEET 16
Introduction . . . . . 8 - 1
Sweet 16: A Psuedo 16 Bit Microprocessor
Description . . . . . 8 - 8
Instruction Descriptions . . . . . 8 - 9
Sweet 16 Opcode Summary . . . . . 8 - 10
Register Instructions . . . . . 8 - 11
Non-Register Instructions. . . . . 8 - 17
Theory of Operation . . . . . 8 - 20
When is a RTS a JSR? . . . . . 8 - 21
Opcode Subroutines . . . . . 8 - 22
Memory Allocation . . . . . 8 - 23
User Modifications . . . . . 8 - 23

APPLESOFT SOURCE LISTING INFORMATION . . . . . 9 - 1

GLOSSARY . . . . . 10 - 1

SAMPLE PROGRAMS
Floating Point Routines . . . . . .11 - 1
Multiply/Divide Routines . . . . . .11 - 1
PRDEC . . . . . .11 - 1
MSGOUT . . . . . .11 - 2
UPCON . . . . . .11 - 2
Game Paddle Printer Driver . . . . . .11 - 2
FORMATTER. . . . . .11 - 4

INTRODUCTION

ASSEMBLY LANGUAGE WHYS AND WHEREFORES

Some of you may ask "What is Assembly Language?" or "Why do I need to use Assembly Language; BASIC suits me fine." While we do not have the space here to do a treatise on the subject, we will attempt to briefly answer the above questions.

Computer languages are often referred to as "high level" or "low level" languages. BASIC, COBOL, FORTRAN and PASCAL are all high level languages. A high level language is one that usually uses English-like words (commands) and may go through several stages of interpretation or compilation before finally being placed in memory. The time this processing takes is the reason BASIC and other high level languages run far slower than an equivalent Assembly Language program. In addition, it normally consumes a great deal more available memory.

From the ground up, your computer understands only two things, off and on. All of its calculations are handled as addition or subtraction but at tremendously high speeds. The only number system it comprehends is Base2 (the Binary System) where a "1" is represented by 00000001 and a "2" is represented by 00000010.

The 6502 microprocessor has five 8-bit registers and one 16-bit register in the ALU (Arithmetic Logic Unit). All data is ultimately handled through these registers. Even this lowest of low-level code requires a program to function correctly. This program is hard wired within the 6502 itself. The microprocessor program functions in three cycles. It fetches an instruction from RAM memory in the computer, decodes it and executes it.

These instructions exist in RAM memory as one, two or three byte groups. (A byte contains 8 binary bits of data and is usually notated in hexadecimal (Base 16) form.) Some early microcomputers allowed data entry only through 16 front panel switches, each of which, when set on or off, would combine in hex. This requires an additional program in the computer to break the byte down into its respective 8 bits so that 6502 may interpret it.

At the next level up (requiring still more programming), the user may enter his/her data in the form of a three character "mnemonic", a type of code whose characters form an association with the microprocessor operation, e.g., LDA stands for "LoaD the Accumulator " . The standard Apple II has a built-in mini-assembler that permits simple Assembly Language programming.

But even this is not sufficient to create a long and comprehensive program. In addition to the use of a three character mnemonic, a full fledged assembler allows the programmer to use "labels", which represent an as yet undefined area of memory where a particular segment of the program will be stored. In addition, an assembler will have a provision for line numbers, similar to those in a BASIC program, which in turn permits the programmer to insert lines into the program and perform other editing operations. And this is what MERLIN is all about.

Finally, a high level language such as BASIC is itself an assembly program which takes a command such as PRINT and reduces it (tokenizes it) to a single hex byte before storing it in memory.

Before using this or any other assembler, the user is expected to be somewhat familiar with the 6502 architecture, modes of addressing, etc. This manual is not intended to teach Assembly Language programming. Many good books on 6502 Assembly programming are available at your local dealer; some are referenced later in this section.

BACKGROUND AND FEATURES

MERLIN is a "Ted-based" editor-assembler. This means that while it is essentially new from the groundup, it adheres to and follows almost all of the conventions associated with TED II+, in terms of the command mnemonics, pseudo-ops, etc.

The original TED ASM was written by Randy Wiggington and Gary Shannon. It has been widely distributed "under the counter" by user groups and individuals, under many names, and in a variety of versions. Seemingly, each person added his own enhancements and improvements. MERLIN is no exception. Representing a major step forward, with the addition of macro capability, MERLIN appears on the scene now as one of the most advanced and sophisticated editor-assemblers for the Apple II, yet retains all of the easy-to-use features of TED that make it desirable to a beginner in assembly language programming.

Significant changes incorporated in MERLIN, in addition to macros, include the use of the logical operators AND, OR, AND EOR, and the math operator for division, the ability to list with or without line numbers, and substantially faster editing. Similarly, the edit module now includes many additional commands to facilitate editing, and the "Read" command allows any Apple text file to be read into the edit buffer, thus permitting the use of source files from other assemblers, such as DOS Tool Kit.

MERLIN assumes that your system has 48K memory and operates on 3.3 DOS. BEWARE of "custom" DOS's. MERLIN does an automatic MAXFILES 2 upon entry, then reverts to the usual value on exit.

SUGGESTED READING

SYSTEM MONITOR - Apple Computer, Inc. Peeking at Call-Apple, Vol I.

APPLE II MINI-ASSEMBLER - Apple Computer Inc. Peeking at Call-Apple Synertek Programming Manual., Synertek 6500-20.

PROGRAMMING THE 6502 - Rodnay Zaks, Sybex C-202.

THE APPLE MONITORS PEELED - WM. E. Dougherty, Apple Computer, Inc.

A HEX ON THEE - Val J. Golding, Peeking at Call-Apple, Vol. II.

FLOATING POINT PACKAGE - Apple Computer, Inc., The Wozpak II

FLOATING POINT LINKAGE ROUTINES - Don Williams, Peeking at Call-Apple Vol I

APPLE II REFERENCE MANUAL - Apple Computer, Inc.

ASSEMBLY LINES - by Roger Wagner
A continuing series of tutorial articles in SOFTALK magazine. The collected series through Volume 1 will be published separately in the near future. An excellent introduction, easy-to-follow for the beginning assembly language programmer.

ASSEMBLY LINES: THE BOOK - by Roger Wagner
A compilation of the first 18 issures of the Assembly lines series. In addition, the text has been extensively edited, and a unique encyclopedia-like appendix added. This appendix shows not only the basic details of each 6502 command, but also a brief discussion of its most common uses, along with concise, illustrative listings.

CONVERTING BRAND X TO BRAND Y - by Randall Hyde
Apple Orchard, Volume 1, No.1, March/April 80. Useful notes and cross references on converting among assemblers.

CONVERTING INTEGER BASIC PROGRAMS TO ASSEMBLY LANGUAGE
by Randall Hyde
Apple Orchard, as above.

HOW TO ENTER CALL - APPLE ASSEMBLY LANGUAGE LISTINGS
Call-APPLE, Volume IV, No.1, January 81.

MACHINE TOOLS
Call-APPLE in Depth, No. 1

SYSTEM REQUIREMENTS

48K APPLE][
16K RAM CARD
80 COLUMN BOARD (optional)
LOWER CASE BOARD (optional)

COMPATIBILITY LIST

VIDEX VIDEOTERM
FULL-VIEW 80 - 80 COLUMN BOARD
M & R SUP'R'TERMINAL 80 COLUMN BOARD
ALS SMARTERM 80 COLUMN BOARD
VISTA VISION 80 - 80 COLUMN BOARD
OMEDA MICROWARE RAMTEX 16 - 16K RAM BOARD
ANDROMEDA 16K BOARD
MICROSOFT 16K RAM BOARD

NOTE: Merlin has been tested with the cards/boards listed above. The author makes no guarantees with respect to the operation of MERLIN with any 80 column boards not listed.

BEGINNERS GUIDE TO USING MERLIN

By T. Petersen

Notes and demonstrations for the beginning MERLIN programmer

INTRODUCTION

The purpose of this section is not to provide instruction in assembly language programming, but to introduce MERLIN to programmers new to assembly language programming in general and MERLIN in particular.

Many of the MERLIN commands and functions are very similar in operation. This section does not attempt to present demonstrations of each and every command option. The objective is to clarify and present examples of the more common operations, sufficient to provide a basis for further independent study on the part of the programmer.

A note of clairification:

Throughout the MERLIN manual, various uses are made of the terms "mode" and "Module".

In this section, "module" refers to a distinct computer program component of the MERLIN system. There are four MODULES:

1. The EXECUTIVE
2. The EDITOR
3. The ASSEMBLER
4. The SYMBOL TABLE GENERATOR

Each module is grouped under one of the two CONTROL MODES:(1) The EXECUTIVE, abbreviated EXEC and indicated by the '$' prompt, or (2) The Editor, indicated by the ':' prompt.

EXECUTIVE CONTROL MODE
Executive Module

EDITOR CONTROL MODE
Editor Module
Assembler Module
SymbolTable Generator Module

The term "mode" may be used to indicate either the current control mode (as indicated by the prompt) or, alternatively: While in control mode and subsequent to the issuance of an entry command, the system is said to be in '[entry command] mode'. For example, while typing in a program after issuing the ADD command, the system is said to be 'in ADD mode'.

Terminating [entry command] mode returns the system to control mode.

INPUT

Programmers familiar with some assembly and higher-level languages will recall the necessity of formatting the input, i.e.: labels, opcodes, operands and comments must be typed in specific fields or they will not be recognized by the assembler program.

In MERLIN, the TABS operator provides a semi-automatic formatting feature.

When entering programs, remember that during assembly, each space in the source code causes a tab to the next tab field. As a demonstration, let's enter the following short routine.

From the very beginning:

1. BRUN MERLIN

2. When the '%' prompt appears at the bottom of the EXEC mode menu, type 'E'. This instantly places the system in EDITOR control mode.

3. As we are entering an entirely new program, use the following entry command after the ':' prompt-type 'A' (for ADD) and press return. A '1' appears one line down and to the right, and the cursor is automatically tabbed one space to the right of the line number. The '1' and all subsequent line numbers which appear after return is pressed, serve roughly the same purpose as line numbers in BASIC, except that in assembly source code, line numbers are not referenced for jumps to subroutines, or in GOTO-like statements.

4. On line 1, enter a '*'. An asterisk as the first character in any line is similar to a REM statement in BASIC - it tells the assembler that this is a remark line and anything after the asterisk is to be ignored. To confirm this, type the title 'DEMO PROGRAM 1' and hit return.

5. After return, the cursor once again drops down one line, a '2' appears and the cursor skips a space.

6. Now, hit the space bar once and type 'OBJ', space again, type '$300', and hit return. Note in most cases the 'OBJ' pseudo-op is neither required nor desirable.

7. On line 3, perform the same sequence: space, type 'ORG', space, type '$300', return.

8. On line 4, do not space once after the line number. Type 'BELL', space, 'EQU' space, '$FBDD', return.

9. Line 5 - Type 'START', space 'JSR', space 'BELL', space, ';' (semicolon), 'RING THE BELL', return. Semicolons are a convention sometimes used within command lines to mark the start of comments.

10. Line 6 - 'END', space, 'RTS', return.

11. The program has been completely entered, but the system is still in ADD mode. To exit ADD just his return, or type Ctrl X, return. The ':' prompt reappears at the left of the screen, indicating that the system has returned to control mode.

12. The screen should now appear like this:

```
1  *DEMO PROGRAM 1
2          OBJ    $300
3          ORO    $300
4  BELL    EQU    $FBDD
5  START   JSR    BELL              ;RING THE BELL
6  END     RTS
```

Note that each string of characters has been moved to a specific field. There are four such fields. Counting right from the column of line numbers:

Field One is reserved for labels. BELL, START and END are examples of labels.

Field Two is reserved for opcodes, such as the MERLIN pseudo-ops OBJ, ORG and EQU, and the 6502 opcodes JSR and RTS.

Field Three is for operands, such as $300, $FBDD and, in this case, BELL.

Field Four will contain any comments.

It should be apparent that the lesson of this exercise is that it is not necessary to input extra spaces in the source file for formatting purposes.

After the line numbers:

Do not space for a label.

Space once after a label (or, if there is no label, once after the line number) for the opcode.

Space once after the opcode for the operand.

Space once after the operand for the comment. If there is no operand, type a space and a semicolon.

SYSTEM AND ENTRY COMMANDS

MERLIN has a powerful and complex built-in editor. Complex in the range of operations possible but, after a little practice, remarkable easy to use.

The following paragraphs contain only minor clairification and brief demonstrations on the use of both sets of commands.

All System and Entry commands are used in EDITOR Control Mode immediately after the ':' prompt.

Ctrl-X, Ctrl-C or a return as the first character of a line exits the current [entry command] mode and returns the system to control mode when ADDing or INSERTing lines. Ctrl-X or Ctrl-C exits edit mode and returns the system to control mode after Editing lines.

The other System and Entry Commands are terminated either automatically or by pressing return.

Inserting and deleting lines in the source code are both simple operations. The following example will INSERT three new lines between the existing lines 4 and 5.

1. After the ':' prompt, type 'I' (INSERT), the number '5', and press return. All inserted lines will preceed the line numbers specified in the command.

2. Input an asterisk, and return. Note that INSERT mode has not been exited.

3. Repeat step 2.

4. Input one space, type 'TYA', and return.

On the screen is now the following

```
:I5
   5*
   6*
   7      TYA
   8
```

5. Hit return, and the system reverts to control mode.

6. LIST the source code.

```
:L
     1  *DEMO PROGRAM 1
     2          OBJ    $300
     3          ORG    $300
     4  BELL    EQU    $FBDD
     5  *
     6  *
     7          TYA
     8  START   JSR    BELL        ;RING THE BELL
     9  END     RTS
```

The three new lines (5,6, and 7) have been inserted , and the subsequent original source lines (now lines 8 and 9) have been renumbered.

Using DELETE is equally easy.

1. In control mode, input 'D5', and return. Nothing new appears on the screen.
2. LIST the source code. The source listing is one line shorter, one of the asterisk-only lines has disappeared, and the subsequent lines have been renumbered.

   It is possible to delete a range of lines in one stop.

1. In control mode, input 'D5,6' and return.
2. LIST the source.

Lines 5 and 6 from the previous example, which contained the remaining asterisk and the TYA opcode, have been deleted, and the subsequent lines renumbered. The listing appears the same as in the subsection on INPUT, Step 13.

This automatic renumbering feature make it important that when deleting lines you remember to begin with highest line number and work back to the lowest.

The Add, Insert, or Edit commands have several sub-commands comprised of Ctrl-characters. To demonstrate, using our BELL routine:

1. After the ':' prompt, enter 'E' (the EDIT command) and a line number ... use '6' for this demonstration, and hit return. One line down and to the right the specified line appears in its formatted state:

```
6 END     RTS
```

and the cursor is over the 'E' in 'END'.

2. Type Ctrl-D. The character under the cursor disappears. Type Ctrl-D again, and a third time, 'END' has been deleted, and the cursor is positioned to the left of the opcode.

3. Hit return and LIST the program. In line 6 of the source code, only the line number and opcode remain.

4. Repeat step 1 (above)

5. This time, type Ctrl-1. Don't move the cursor with the space bar or arrow keys. Type the word 'END', and return.

6. LIST the program. Line 6 has been restored.

If you are editing a single line, hitting return restores the control mode prompt. In step 1 (above), if you had specified a range of lines (ex:'E3,6') after issuing the EDIT command, return would have called up the next sequential line number within the specified range. As the lines appear, you have the options of editing using the various sub-commands, pressing return which will call up the next line, or exiting the EDIT mode using Ctrl-C. Note hitting return will enter the entire line in memory, exactly as it appears on the screen.

The other sub-commands, Ctrl-characters used under the EDIT command, function similarly, Read the definitions in Section 3 and practice a few operations.

ASSEMBLY

The next step in using MERLIN is to assemble the source code into object code.

After the ':' prompt, type the edit module system command ASM and hit return. On your screen is now the following;

UPDATE SOURCE (Y/N)?

Type N, and you will see:

```
ASM                     1           *DEMO PROGRAM   1
                        2                   OBJ     $300
                        3                   ORG     $300
                        4           BELL    EQU     $FBDD
0300    °  20 DD FB     5           *START  JSR     BELL
           ;RING THE BELL
0303    60              6           END     RTS

--END ASSEMBLY--

ERRORS:  0

4 BYTES

SYMBOL TABLE - ALPHABETICAL ORDER

      BELL      =$FBDD          ?     END             =$0303

?     START     =$0300

SYMBOL TABLE - NUMERICAL ORDER

?     START     =$0300          ?     END             =$0303

      BELL      =$FBDD
```

If instead of completing the above listing, the system beeps and displays an error message, note the line number referenced in the message, then press return until the "...BYTES..." message appears. Then refer back to the subsection on INPUT and compare the listing with step 13. Look especially for elements in incorrect fields.

If all went well, to the right of the column of numbers down the middle of the screen is the, by now familiar, formatted source code.

To the left of the numbers, beginning on line 5,is a series of numeric and alphabetic characters. This is the object code - the opcodes and operands assembled to their machine language hexadecimal equivalents.

Left to right, the first group of characters is the routine's starting address in memory (see the definition of OBJ and ORG in Section 3 - MERLIN System Pseudo-Ops). After the colon is the number '20'. This is the one-byte hexadecimal code for the opcode JSR.

Note that the label 'START' is not assembled into object code; neither are comments, remarks, or pseudo-ops such as OBJ and ORG. Such elements are only for the convenience and utility of the programmer and the use of the assembler program. They are of no use to the computer and, therefore, are not translated into the machine's language.

The next two bytes (each pair or characters is one byte) on line 5 bear a curious resemblance to the last group of characters on line 4; have a look. In line 4 of the source code we told the assembler that the label 'BELL' EQUated with address $FBDD. In line 5, when the assembler encountered 'BELL' as the operand, it substituted the specified address. The sequence of the high and low-order bytes was reversed, a 6502 microprocessor convention.

The rest of the information presented explains itself: The total errors encountered in the source code was zero, and four bytes of object code (count the bytes following the addresses) was generated.

SAVING AND RUNNING PROGRAMS

The final step in using MERLIN is running the program. Before that, it would be a good idea to save the source code.

1. Return to control mode, if necessary, and type 'Q', return. the system has quit EDITOR Mode and reverted to EXECUTIVE (EXEC) mode. If the MERLIN Systemdisk is still in the drive, remove it and insert an initialized work disk.

   After the 'Z' prompt, type 'S' (the EXEC mode SAVE SOURCE FILE command). The system will ask for a filename. Type 'DEMO1', return. After the program has been saved, the prompt returns.

2. Type 'C' (CATALOG) and scan the disk catalog. The source code has been saved as a binary file titled "DEMO1.S". The suffix ".S" is a file-labelling convention which indicates the subject file is source code. This suffix is automatically appended to the name by the S (SAVE SOURCE FILE) command.

3. Hit <RETURN> to return to EXEC mode and input 'O', for OBJECT CODE SAVE. The object file is saved under the same name as was earlier specified for the source file. There is no danger of overwriting the source file because no suffix is appended to object code file names.

OBJECT CODE SAVE must be preceded by a successful assembly.

While writing either file to disk, MERLIN also displays the address parameter, and calculates and displays the length parameter. It's a good practice to take note of these. Viewing the catalog will show that although the optional A$ and L$ parameters were displayed on the EXEC mode menu, they were not saved as part of the file names. If you'd prefer to have this information in the disk catalog, use the DOS RENAME command. Make sure no commas are included in the new file name.

Return to EDITOR mode, type 'MON', return, and the monitor prompt '*' appears. Input '300G'. Press return. A beep is heard. the demonstration program was responsible for it. It works!

Now you can return to the EXEC by typing Ctrl-Y and hitting return . A control character is always entered by holding down the CTRL key and the character key together.

MAKING BACK-UP COPIES OF MERLIN

To make back-up copies of the MERLIN diskette, you should use the copy program provided on the original diskette. To run this, simply boot on the diskette, and as the drive light comes on, press the 'C' key on the keyboard. The program will verify your intention to produce a copy, and then ask the slot and drive values for the SINGLE DRIVE on which the copy is to be made. A a single drive is used to assure maximum reliability in the copy.

Be careful during the copy process to appropriately alternate the original and copy diskettes. If at any time, you make an error, STOP IMMEDIATELY and re-boot to start over.

When completed, it is highly recommended that you use ONLY the BACK-UP copy of MERLIN in your daily work, and keep the original in a safe place. The copy program will provide a total of 3 back-up copies, giving you a total of 4 working copies of the MERLIN assembler. In addition, files may be transferred from one diskette to another. This means that damaged files on a given diskette may be replaced with a known "good" file from another diskette. Use the FID file transfer utility on your Apple System Master diskette to move the file from the working diskette to the one on which the file was damaged. All library files, and also the side containing SOURCEROR.FP can be moved to any DOS 3.3 diskette using the FID utility program.

EXECUTIVE MODE

C:CATALOG

After showing the catalog, this command accepts any disk command you wish to give, using standard DOS syntax. Unlike the Load, Append and Save commands, you must type the .S suffix for a source file. This facility is provided primarily for locking and unlocking files. Do not use it to load or save files. If you do not want to give a disk command, just hit return. To cancel a partially typed command use control X or make sure the command is in the wrong syntax (type some commas) or just backspace to the beginning. If you type Control-C <return> after "COMMAND:", you will be presented with the EXEC mode prompt "]" and then you may issue any exec command such as L for load. This permits you to give an EXEC mode command while the catalog is still on the screen. In addition, if Control-C is typed at the "CATALOG pause" point, printing of the remainder of the catalog is aborted.

L:LOAD

This is used to load a source file from disk. You will be asked for the name of the file. You should not append a .S, since MERLIN does this automatically. If you have hit L by mistake, just hit <RTN> twice and the command will be cancelled without affecting any file that may be in memory. After a load(or append) command, you are automatically placed in editor mode, just as if you hit E. The source will automatically be loaded to the correct address. Subsequently LOAD or SAVE commands will display the last used file name, followed by a flashing "?". If you hit the "Y" key, the current file name will be used for the command. If you hit any other key (e.g., [RETURN],) the cursor will be placed on the first character of the file name, and you may type in the desired name. [RETURN] alone at this time will cancel the command.

S:SAVE

Use this to save a source file to disk. As in the load command, you do not specify the suffix .S, and you can hit <RTN> to cancel the command. Note that the address and length of the source file are shown on the MENU.

These are for information only. You should not use these for saving; the assembler remembers them better than you can, and sends them to DOS automatically. As in the LOAD command above, the file name will be displayed and you may type "Y" to SAVE the same file name, or any key for a new file name.

A:APPEND

This loads in a specified source file and places it at the end of the file currently in memory. It operates in the same way as the load command, and does not affect the default file name. It does not save the appended file; you are free to do that if you wish.

D:DRIVE

When you hit D, the drive used for saving and loading changes from 1 to 2 or 2 to 1. The currently specified drive is shown on the menu. When MERLIN is first BRUN, the specified drive will be the one used for the BRUN. There is no command to specify slot number, but this can be accomplished by typing C for catalog, then giving the disk command CATALOG, Sn, where n is the slot number.

E:EDITOR

This command places you in the editor/assembler mode. It automatically sets the default tabs for the editor to those appropriate for source files.

O:SAVE OBJECT CODE

You are permitted to use this command only after the successful assembly of a source file. In this case you will see the address and length of the object code on the menu. As with the source address, this is given for information only. Note that the object address shown is that of the program's ORG (or $8000 by default) and not that of the actual current location of the assembled code (which is $8000 or whatever OBJ you have used). When you use this command, you are asked for a name for the object file. Unlike the source file case, no suffix will be appended to this name.

Thus you can safely use the same name as that of the source file (without the .S of course). When this object code is saved to the disk, its address will be the correct one, the one shown on the menu. When, later, you BLOAD it or BRUN it, it will go to that address, which can be anything ($300,$800, etc.) Thus there is usually no need to use an OBJ in the source code, unless the object code will be too long for the space available at $8000 and above.

Q:QUIT

This exits to BASIC. You may reenter MERLIN by issuing the "ASSEM" COMMAND. This reentry will be a warm start, that is, it will not destroy the source file currently in memory. This exit can be used to give disk commands, if that is more convenient than the one provided by C.

R:READ

This reads text files into MERLIN. It is always an "append". Thus type "NEW" in the editor if you don't want an append. If no file is in memory then the name given will become the default file name. Appended reads will not do this. When the read is complete, you are placed in the editor. If the file contains lines longer than 255 characters, these will be divided into two or more lines by the READ routine. The file will be read only until it reaches HIMEM and will produce a memory error if it goes beyond. The READ (and the WRITE) command will append a "T." to the beginning of the file name you specify UNLESS you precede the file name with a space (or any character less than "@"; this character will not be sent to DOS).

W:WRITE

This writes a MERLIN file into a text file instead of a binary file. The speed of the READ and WRITE routines is approximately that of a BLOAD or BSAVE. The WRITE routine does a VERIFY after the write.

THE EDITOR

Command Mode

Basically there are three modes in the editor: the command mode, the add or insert mode, and the edit mode. The main one is the command mode, which has a colon : as prompt.

For many of the command mode commands, only the first letter of the command is required, the rest being optional. We show the required command characters in upper case and the optional ones in lower case. In some commands, you must specify a line number, a range or a range list. A line number is just a number. A range is a pair of line numbers separated by a comma. A range list consists of several ranges separated by slashes, /.

Several commands allow specification of a string. The string must be "delimited" by a non-numeric character other than the slash, /. Such a delimited string is called a d-string. The usual delimiter is single or double quote marks,',".

Line numbers in the editor are provided automatically. You never type them when entering text; only when giving commands. If a line number in a range exceeds the number of the last line, it is automatically adjusted to the last line number. The commands are:

HImem: (decimal number between the end of source and 38995=$9853) This command is rarely needed. It sets the upper limit for the source file. Himem defaults to $8000, and so does not have to be set unless you use a non-default object address.

NEW

Deletes present source file, resets Himem to $8000 and starts fresh.

PR# (0-7)

Same function as in BASIC. Mainly used for sending an editor or assembly listing to a printer. DO NOT use this to select an 80-column card.

USER

This does a JSR $3F5. (That is the Applesoft ampersand vector location, which normally points to an RTS.) The designed purpose of this command is for the connection of user defined printer drivers. (You must be careful that your printer driver does not use zero page addresses, except the I/O pointers and $60 - $6F because this will likely interfere with MERLIN's heavy zero page usage.)

TABS number, number, ...
TABS number, number, ..., "tab character"

This sets the tabs for the editor, and has no effect on the assembler listing. Up to 9 tabs are possible. The default tab character is a space, but any may be specified. The assembler regards the space as the only acceptable tab character for the separation of labels, opcodes, and operands. If you don't specify the tab character, then the last one used remains. Entering TABS and a carriage return will set all tabs to zero.

LENgth

This gives the length in bytes of the source file, and the number of bytes remaining before Himem (usually $8000 - not BASIC Himem).

Where (line number)

This prints in hex, the location in memory of the start of the specified line. Where 0 (or "W0") will give the location of the end of source.

MONitor

This exits to the monitor. Thus you may reenter by either Control-C, Control-B or control Y. These reestablish the important zero page pointers from a save area inside MERLIN itself. Thus control Y will give a correct entry, even if you have messed up the zero page pointers while in the monitor. DOS is not connected when using this entry to the monitor. This facility is designed for experienced Apple programmers, and is not recommended to beginners. You may reenter the editor directly with a OG.

This reentry, unlike the others, will use the zero page pointers at $A - $F instead of the ones saved upon exit, so you must be sure that they have been altered.

TRuncON

This sets a flag which, during LIST or PRINT, will terminate printing of a line upon finding a space followed by a semicolon. It makes reading of source files easier on the Apple 40 column screen. In the assembler, this limits printing of the object code to 3 bytes per line and has no effect on comments.

TRuncOFf

This returns to the default condition of the truncation flag. (This also happens automatically upon entry to the editor from the exec mode or from the assembler.) In the assembler, this directs that all object bytes be printed.

Quit

Exits to exec mode.

ASM

This passes control to the assembler, which attempts to assemble the source file. First, however, you are asked if you wish to "update the source". This is to remind you to change the date or identification number in your source file. If you answer "N" then the assembly will proceed. If you answer "Y" then you will be presented with the first line in the source which contains a "/" and are placed in edit mode. When you are doneediting this line and hit return, assembly will begin. If you use the control C edit abort command, however, you will return to the editor command mode, and any I/O hooks you have established, by PR# etc. will have been disconnected. This will also happen if there is no line with a "/".

Note that by establishing a comment line with "*/" at the beginning, you have a nearly automatic method of keeping track of multiple versions of a program.

Control-D

During the second pass of assembly, typing a Control-D will toggle the list flag, so that listing will either stop or resume. this will be defeated if a LST opcode occurs in the source, but another Control-D will reinstate it.

Delete (line number)
Delete (range)
Delete (range list)

This deletes the specified lines. Since, unlike BASIC, the line numbers are fictitious, they change with any insertion or deletion. Thus you must specify the 'higher' range first!

Replace (line number)
Replace (range)

This deletes the line number or range, then places you into insert mode at that location.

List
List (line number)
List (range)
List (range list)

Lists the source file with added line numbers. Control characters in source are shown in inverse, unless the listing is being sent to a printer or other nonstandard outport. The listing can be aborted by control C or with key "/". You may stop the listing by hitting the space bar and then advance a line at a time by hitting the space bar again. Any other key will restart it. This space bar pause also works during assembly and the symbol table print out.

(period)

Lists starting from the beginning of the last specified range. For example, if you type L10,100 then lines 10 to 100 will be listed. If you then use ".", listing will start again at 10 and continue until stopped (the end of the range is not remembered).

/

/ (line number)

This continues listing from the last line number listed, or, when a line number is specified, from that line. This listing continues to the end of the file or until it is stopped as in List.

Print

Print (line number)

Print (range)

Print (range list).

This is the same as "list" except that line numbers are not added.

PRinTeR command

This command is for sending a listing to a printer with page headers and provision for page boundary skips. The default parameters may be set up using the configuration program. The syntax of this is:

PRTR slot# "string to initialize printer" page header desired"

If the slot number used is more than 7 then instead a JSR $3F5 (ampersand vector) is done and it is expected that the routine there will connect a printer driver by putting its address $36-$37.

The page header can be omitted and the header will then consist of page numbers only.

The initialization string may not be omitted. -If no special string is required by the printer, then use some unrecognized control character or a null string (in which case a carriage return will be used). Examples of initialization strings are <Control-Q> for IDS printers, or <Control-I> 80N for Apple cards.

PRTR 0 (no strings required here) will allow you to see where the page breaks occur. If an 80 column card is in use in slot 3, then use PRTR 3 for this. No output is sent to the printer until a list, print, or ASM command is issued.

Find (d-string)
Find (range) (d-string)
Find (range list) (d-string)

This lists those lines containing the specified string. It may be aborted with control C or key "/". Since the control L case toggle works in command mode, you can use it to find or change strings with lower case characters.

Change (d-string d-string)
Change (range) (d-string d-string)
Change (range list) (d-string d-string)

This changes occurrences of the first d-string to the second d-string. The d-strings must have the same delimiter with the adjoining ones coalescing. For example, to change occurrences of "speling" to "spelling" throughout the range 10,100, you would type C20,100"speling" spelling". If no range is specified, the entire source file is used. Before the change operation begins, you are asked whether you want to change "all" or "some". If you select "some" by hitting the "S" key, the editor stops whenever the first string is found and displays the line as it would appear with the change. If you then hit <ESCAPE> or any control character, the change displayed will not be made. Any other key, such as the space bar, will accept the change. Control C or key "/" will abort the change process.

COPY (range) TO(line number)
COPY (line number)TO(line number)

This copies the range to just 'above' the specified number. It does not delete anything.

MOVE (range) TO (line number)

MOVE (line number) TO (line number)

This is the same as COPY but, after copying, automatically deletes the original range. You always end up with the same lines as before, but in a different order.

Edit
Edit (line number)
Edit (range)
Edit (range list)
Edit (d-string)
Edit (line number) (d-string)
Edit (range) (d-string)
Edit (range list) (d-string)

This presents the range, etc., line by line to be edited and puts you into the edit mode. If a d-string is appended, then only those lines containing the d-string are presented.

Hex-Dec Conversion

If, in the command mode, you type a decimal number (positive or negative) then the hex equivalent is returned. If you type a hex number, prefixed by "$", the decimal equivalent is returned. All commands accept hex numbers, which is mainly convenient for the HIMEM and SYM commands.

TEXT

This converts ALL spaces in a source file to inverse spaces. The purpose is for use on "text" files so that it is not necessary to remember to zero the tabs before printing such a file. This conversion has no effect on anything except the editor's tabulation.

FIX

This undoes the effect of TEXT. It also does a number of technical housekeeping chores. It is recommended that the command FIX to be used on all files from external sources, after which the file should be saved. (Note that the TEXT and FIX routines are written in SWEET 16 and thus are somewhat slow. Several minutes may be needed for their execution on large files.) FIX or an edit will truncate any lines longer than 255 characters.

SYM

MERLIN places the symbol table on the language card (in bank 1 of $D000-$DFFF). This space is quite adequate for all but gigantic programs. In case this space is used up, however, the SYM command gives you a means to direct the assembler to continue the symbol table in another area. If you type SYM $9000, for example, and assemble the program, then when and if the symbol table uses up its normal space, it will be continued at $9000 until BASIC HIMEM. It must be noted that the SYM command will be cancelled by a HIMEM command or by exit to EXEC mode and reentry (Thus, set HIMEM before setting up a SYM address.) The SYM address must be above HIMEM and below BASIC HIMEM. If the symbol table grows beyond the allotted space, than you will get a memory error during the first pass of assembly.

VIDeo

This command is designed to select or deselect an 80 column board. The default condition can be selected using the Configuration program. This is similar to use of PR# in basic. DO NOT USE PR# to select an 80 column board, MERLIN will definitely not like it. PR# is designed for selection of a printer ONLY. An 80 column board in slot 3 for example, can be selected by typing, from the editor: VIDEO 3. It is deselected by VIDEO 0 or VIDEO $10 possibly followed by [RESET]. The latter two forms both select the standard Apple screen, but VIDEO 0 will cause all lower case output to the screen to be converted to upper case except for lower case in the source file will be converted to flashing upper case. (Output to a printer is never converted.) Thus, if you have a lower case adaptor, you will want to use VIDEO $10 (or VIDEO 16) instead of VIDEO 0 when selectng the Apple screen. If your 80 column card has a software screen switch via an escape sequence, this may be used to return to 40 column mode. This will be equivalent to "VID $10" and would have to be followed by a VID 0 if you don't have a lower case adapter. (For example, use [ESC] Control-Q [RETURN] on the Smarterm or [ESC] Q Control-X on the Sup"R"term.)

FW"word"
(Find Word) This is an alternative to the (F)ind command. It will find the specified word only if it is surrounded, in source, by non-alphanumeric characters. Thus FW"CAT" will find:

```
CAT
CAT-1
(CAT,X)
```

but will not find CATALOG or SCAT.

CW "word"new stuff"
(Change Word) This works as described under FW.

EW "word"
(Edit word) This is to edit as FW is to find.

VAL "expression"
This will return the value of the expression as the assembler would compute it.
Examples:

VAL "LABEL"	gives the address (or value) of LABEL for the last assembly done or " unknown label" if not found.
VAL "$1000/2"	returns $0800
VAL "%1000"	returns $0008
VAL !"A" - "0"!	returns $0011

NOTE: for the commands involving a d-string, the character "^" acts as a "WILD CARD". Thus F"Jon^s" will find both "Jones" and "Jonas".

Add/Insert Mode

The Add and Insert modes in the editor act as if you are in the mode, except that Control-R will do nothing and the exit from Add mode acts as described. Thus hitting "Return", for example, will accept the entire line as shown on the screen.

Add

This places you into the "Add" mode. This acts much like entering BASIC lines with auto line numbering. However, you may enter lower case text (useful for comments if you have a lower case adapter) by typing control L. This acts as a case toggle, so another control L returns you to uppercase mode. Also the exit from "Add" mode is to just hit RETURN as the FIRST character of a line. You may enter an EMPTY line by typing a space and then RETURN. (This will not enter the space into text, it only bypasses the exit. The editor automatically removes extra spaces at the end of lines.) You may also exit the "Add" mode by control X or control-C, which then cancels the current line.

Insert (line number)

This allows you to enter text just 'above' the specified line. Otherwise, it functions the same as "Add" mode.

Control-L

Toggles the current case. If you are in upper case, Ctrl L will place you in lower, and vice versa. Upper case is defaulted to when entering each new line.

To change the case of a word, type Control-L and then copy over the word using the right arrow.

Edit Mode

After typing E in the editor, you are placed in edit mode. The first line of the range you have specified is placed on the screen with the cursor on its first character. The line is tabbed as it is in listing, and the cursor will jump across the tabs as you move it with the arrow keys. When you are through editing, hit RETURN.

The line will be accepted as it appears on the screen, no matter where the cursor is when you hit RETURN. The edit commands and functions are very similar, but not identical, to those in Neil Konzen's Program Line Editor.

The Edit Mode Commands are (all commands except Control-R are available in Add and Insert modes.):

Control-I
Begins insertion of characters. This is terminated by any control character except the control L case toggle, such as the arrows or return.

Control-D
Deletes the character under the cursor.

Control-F
Finds the next occurrence of the character typed after the control F. This is recursive.

Control-O
Functions as Control I, except inserts any control character (including the command characters such as control Q).

Besides enabling the insertion of control characters, Control-O also allows the user to type characters not normally available on the Apple keyboard. On the following page is a table of the conversions:

Control-O followed by:

<	gives	Control _
>	"	Control \
K	"	[
L	"	\
M	"	]
N	"	^
O	"	
k	"	_
l	"	{
m	"	|
n	"	}
o	"	~
		(whatever $FF gives on your machine)

(Note that if you are using a shift key modification then, depending on which one you have, shift M may give upper case M and you will have to use Control O to get the right bracket.)

Control-P
If entered as first character of a line gives 32 *'s.

Control-@
If entered as first character of a line gives 30 spaces bordered by *'s.

Control-C or Control-X
Aborts edit mode and returns to the editor's warm start. The current line being edited will retain its original form.

Control-B
Places the cursor at the beginning of the line.

Control-N
Places the cursor one space to the right of the end of the line.

Control-R
Returns the line to its original form. (Not available in Add and Insert modes.)

Control-Q
Deletes the part of the line following the cursor.

[Return]
Accepts the line as it appears on the screen and fetches the next line to be edited, or goes to the warm start if the specified range has been completed.

The editor automatically replaces spaces in comments and ASCII strings with inverse spaces. When listing, it converts them back, so you never notice this. The reason for this is to avoid inappropriate tabbing of comments and ASCII strings.

In the case of ASCII strings, this is only done when the delimiter is a quote or a single quote. You can, however, accomplish the same thing by editing the line, replacing the first delimiter with a quote, hitting RETURN, and then editing again and changing the delimiter back to the desired one.

THE ASSEMBLER

This documentation will not attempt to teach you assembly language. It will only explain the syntax you are expected to use in your source files, and to document the features that are available to you in the assembler.

Number format

This assembler accepts decimal, hexadecimal, and binary numerical data. Hex numbers must be preceded by "$" and binary numbers must be preceded by "%", thus the following four instructions are all equivalent:

LDA #100 LDA #$64 LDA #%1100100 LDA #%01100100

(As indicated, leading zeros are ignored.) The "#" here stands for "number" or "data", and the effect of these instructions is to load the accumulator with the number (decimal) 100.

A number not preceded by "#" is interpreted as an address. Thus;

LDA 1000 LDA $3E8 LDA %1111101000

are equivalent ways of loading the accumulator with the byte that resides in memory location $3E8.

Use the number format that is appropriate for clarity. For example, the data table:

```
DA    $1
DA    $A
DA    $64
DA    $3E8
DA    $2710
```

is a good deal more mysterious that its decimal equivalent:

```
DA    1
DA    10
DA    100
DA    1000
DA    10000
```

Source Code Format

A line of source code typically looks like:

LABEL OPCODE OPERAND ;COMMENT

A line containing only a comment must begin with a "*". Comment lines starting with ";" are accepted and tabbed to the comment field. The assembler will accept an empty line in the source code and will treat it just as a SKP 1 instruction (see the section on pseudo opcodes), except the line number will be printed.

The number of spaces separating the fields is not important, except for the editor's listing, which expects just one space.

The maximum allowable LABEL length is 13 characters, but more than 8 will produce messy assembly listings. A label must begin with a character at least as large, in ASCII value, as the colon, and may not contain any characters less, in ASCII value, than the number zero.

The assembler examines only the first 3 characters of the OPCODE (with certain exceptions such as the Sweet 16 opcode POPD) so, for example, you can use PAGE instead of PAG.(Because of the exception, the fourth letter should not be a D, however.) The assembler listing will truncate the opcode to 7 letters and will not look well with one longer than 4 unless there is no operand.

The maximum allowable combined OPERAND+COMMENT length is 64 characters. You will get an error if you use more than this. A comment line by itself is also limited to 64 characters.

Expressions

To make clear the syntax accepted and/or required by the assembler, we must define what is meant by an "expression". Expressions are built up from "primitive expressions" by use of arithmetic and logical operations. The primitive expressions are:

1. A label
2. A decimal number
3. A hexadecimal number (preceded by "$")
4. A binary number (preceded by "%")
5. Any ASCII character preceded, or enclosed, by quotes or single quotes
6. The character * (standing for the present address)

All number formats accept 16 bit data and leading zeros are never required. In case 5 the "value" of the primitive expression is just the ASCII value of the character. The high bit will be on if a quote ["] is used, and off if a single quote ['] is used.

The assembler supports the four arithmetic operations : +, -, /, and *. It also supports the three logical operations:

! = Exclusive OR . = OR & = AND

Thus some examples of legal expressions are:

```
LABEL1-LABEL2
2*LABEL+$231
1234+%10111
"K"-"A"+1
"0"!LABEL
LABEL&$7F
*-2
LABEL.%10000000
```

Parentheses have another meaning and are not allowed in expressions. All arithmetic and logical operations are done from left to right. (Thus 2+3*5 would assemble as 25 and not 17.)

Immediate Data

For those opcodes such as LDA, CMP, etc., which accept immediate data (numbers as distinct from addresses) the immediate mode is signalled by preceding the expression by "#". An example is: LDX #3. In addition:

#<expression	produces the low byte of the expression
#>expression	produces the high byte of the expression
#expression	also gives the low byte (the6502 does not accept 2-byte DATA)
#/expression	is optional syntax for the high byte of the expression

The ability of the assembler to evaluate expressions such as LAB1-LA2-1 is very useful for the following type of code:

```
COMPARE     LDX     #FOUND-DATA-1
LOOP        CMP     DATA,X
            BEQ     FOUND
            DEX
            BPL     LOOP
            JMP     REJECT    ;not found
DATA        HEX     E3BC3498
FOUND       RTS
```

With this type of code, if you add or delete some of the "Data", then the appropriate X-index for the comparison loop is automatically adjusted.

Addressing Modes

6502 OPCODES

The assembler accepts, of course, all the 6502 opcodes with standard mnemonics. It also accepts BLT (branch if less than) as an equivalent to BCC and BGE (branch if greater or equal) as an equivalent to BCS.

There are 12 addressing modes on the 6502. The appropriate syntax for these, in MERLIN are:

Addressing mode	Syntax	Examples
Implied	OPCODE	CLC
Accumulator	OPCODE	ROR
Immediate (data)	OPCODE#expr	ADC #$F8
		CMP #"M"
		LDX #>LABEL1-LABEL2-1
Zero page (address)	OPCODE expr	ROL 6
Indexed X	OPCODE expr,X	LDA $E0,X
Indexed Y	OPCODE expr,Y	STX LAB,Y
Absolute (address)	OPCODE expr	BIT $300
Indexed X	OPCODE expr,X	STA $4000,X
Indexed Y	OPCODE expr,y	SBC LABEL-1,Y
Indirect	JMP (expr)	JMP ($3F2)
Preindexed X	OPCODE (expr,X)	LDA (6,X)
Postindexed Y	OPCODE (expr),Y	STA ($FE),Y

Note that there is no difference in syntax for zero page and absolute modes. The assembler automatically uses zero page mode when appropriate. In the indexed, indirect modes, only a zero page expression is allowed, and the assembler will give an error message if the "expr" does not evaluate to a zero page address.

Note also that the "accumulator mode" does not require (or accept) an operand. Some assemblers perversely require you to put an "A" in the operand for this mode.

The assembler will decide the legality of the addressing mode for any given opcode.

Additionally, Merlin provides the ability to FORCE non-zero page addressing. The way to do this is to add anything (except "D") to the end of the opcode.
Example:

```
LDA  $10   assembles as zero page (2 bytes) while,
LDA: $10   assembles as non-zero page (3 bytes).
```

SWEET 16 OPCODES

The assembler also accepts all Sweet 16 opcodes with the standard mnemonics. The usual Sweet 16 registers R0 to R15 do not have to be "equated" and the "R" is optional. TED II+ users will be glad to know that the SET opcode works as it should, with numbers or labels. For the SET opcode, either a space or a comma may be used between the register and the data part of the operands; that is, SET R3,LABEL is equivalent to SET R3 LABEL. It should be noted that the NUL opcode is assembled as a one-byte opcode (the same as HEX 0D) and not a two byte skip as this would be interpreted by ROM Sweet 16. This is intentional, and is done for internal reasons.

PSEUDO OPCODES

Directives

EQU expression (EQUals)
= expression (optional syntax)

Used to define the value of a LABEL, usually an exterior address or an often used constant for which a meaningful name is desired. It is recommended that these all be located at the beginning of the program. The assembler will not permit an "equate" to a zero page number after the label equated has been used, since bad code could result from such a situation. (Also see "Variables".)

ORG expression (ORiGin)

Establishes the address at which the program is designed to run. It defaults to the present value of HIMEM ($8000 by default). Ordinarily there will be only one ORG and it will be at the start of the program. If more than one ORG is used, the first one establishes the BLOAD address. This can be used to create an object file that would load to one address but designed to run at another address. You cannot use ORG *-1 (etc) to back up the object pointers as is done in some assemblers. This must be done instead by DS -1.

OBJ expression (OBJect)

Establishes the address at which the object code will be placed during assembly. It defaults to HIMEM. There is rarely any need to use this pseudo-op and inexperienced programmers are urged not to use it. An OBJ above BASIC HIMEM (or the SYM address, if any) will defeat generation of object code. This may be used when sending a long listing to a printer or when using direct assembly to disk (opcode DSK).

PUT filename

PUT FILENAME, (drive and slot parameters accepted in standard DOS syntax) will read the named file (with the "T." prefix appended unless the filename starts with a character less than "@") and "inserts" it at the location of the opcode. Note - "insert" refers to the effect on assembly of the location of the source. The file itself is actually placed just following the main source.) Text files are required by this facility in order to insure memory protection. A memory error will occur if a PUT file goes beyond HIMEM. These files are in memory only one a time, so a very large program can be assembled using the PUT facility.

There are two restrictions on a PUT file. First, there cannot be macro DEFINITIONS inside a PUT file (they must be in the main source). Second, a PUT file may not call another PUT file with the PUT opcode. Of course, linking can be simulated by having the "main program" just contain the macro definitions and call, in turn, all the others by the PUT opcode. Any variables (e.g.,]LABEL) may be used as "local" variables. The usual local variables]1 through]8 may be set up for this purpose using the VAR opcode.

The PUT facility provides a simple way to incorporate much used subroutines, such as MSGOUT or PRDEC, in a program.

VAR expr; expr; expr...

This is just a convenient way to equate the variables]1 -]8. "VAR 3;$42;LABEL" will set]1 = 3,]2 = $42, and]3 = LABEL. This is designed for use just prior to a PUT. If a PUT file uses]1 -]8 (execpt in >>> lines for calling macros) then there MUST be a previous declaration of these.

SAV filename

SAVE FILENAME, (drive and slot parameters accepted) will save the current object code under the specified name. This acts exactly as does the EXEC mode object saving command, but it canbe done several times during assembly.

This pseudo-opcode provides a means of saving portions of a program having more than one ORG. It also enables the assembly of extremely large files. After a save, the object address is reset to the last specification of OBJ or to HIMEM by default.

The SAVE command sets the address of the saved file to its correct value. Thus, for example, if your program contains three SAV commands, then it will be saved in three pieces. When BLOADed later, they will go to the correct locations, the third following the second and that following the first.

Together, the PUT and SAV opcodes make it possible to assemble extremely large files.

DSK filename

DSK FILENAME will direct the assembler to assemble the following code directly to disk. If DSK is in effect, then the old file will be closed and the new one begun. This is useful primarily for extremely large files. For moderately sized programs, .SAV is to be preferred since it is 30% faster and theoretically more reliable. Because of the way it works, the CHK opcode is incompatible with DSK and will be disabled if DSK is in effect.

END

This rarely used or needed pseudo opcodeinstructs the assembler to ignore the rest of the source. Labels occurring after END will not be recognized.

Formatting

LST ON or OFF (List)

This controls whether the assembly listing is to be sent to the Apple screen and/or other output device. You may, for example, use this to send only a portion of the assembly listing to your printer. Any number of LST instructions may be in the source. If the LST condition is OFF at the end of assembly, then the symbol table will not be printed. The assembler actually only checks the third character of the operand to see whether or not it is a space.

Thus, LST ERINE will have the same effect as LST OFF. The LST directive will have no effect on the actual generation of object code. If the LST condition is OFF, then the object code will be generated much faster, but this is recommended only for debugged programs. Note that Control-D from the keyboard toggles this flag during the second pass.

EXP ON or OFF (EXPand)

EXP ON will print an entire macro during the assembly. The OFF condition will print only the PMC pseudo-op. EXP defaults to ON. This has no effect on the object coded generated.

PAU (PAUse)

On the second pass this causes assembly to pause until a key is hit. (This can also be done from the keyboard by hitting the space bar.)

PAG (PAGe)

This sends a formfeed ($8C) to the printer. It has no effect on the screen listing even when using an 80-column card.

AST expression (ASTerisks)

This sends Asterisks to the listing, the same number as the value of the operand. The number format is the usual one, so that AST 10 will send (decimal) 10 asterisks, for example. The number is treated modulo 256 with 0 being 256 asterisks! This differs from TED II+, which recognizes the operand as a hex expression, thus will need to be converted.

SKP expression (SKiP)

This sends OPERAND number of carriage returns to the listing. The number format is the same as in AST.

TR opcode

This has the same effect in the assembler as does the Editor TR Command. Thus, TR or TR ON limits object code printout to 3 bytes per line and TR OFF resets it to print all object bytes.

Strings

ASC d-string (ASCii)

This puts a delimited ASCII string into the object code. The only restriction on the delimiter is that it does not occur in the string itself. Different delimiters have different effects. Any delimiter less than (in ASCII code) the single quote ['] will produce a string with the high bits on, otherwise the high bits will be off. Thus, for example, the delimiters !"#$%& will produce a string in "negative" ASCII, and the delimiters '()+? will produce one in "positive" ASCII. Usually the quote and single quote are the delimiters of choice, but other delimiters provide the means of inserting a string containing the quote or single quote as part of the string.

DCI d-string (Dextral Character Inverted)

This is the same as ASC except that the string is put into memory with the last character having the opposite high bit to the others.

INV d-string (INVerse)

This puts a delimited string in memory in inverse format. All choices of delimiter have the same effect.

FLS d-string (FLaSh)

This puts a delimited sting in memory in flashing format. All choices of delimiter have the same effect.

REV d-string (REVerse)

Example: REV "DISK VOLUME" gives;

EMULOV KSID (delimiter choice as in ASC)

Data and Allocation

DA expression (Define Address)

This stores the (two byte) value of the operand, usually an address, in the object code, low byte first. DA $FDF0 will generate F0 FD. Also accepts multiple data (e.g DA 1,10,100)

DDB expression (Define Double Byte)

As above, but places high byte first. Also accepts multiple data (e.g. DDB 1,10,100)

DFB expression (DeFine Bytes)

This puts the bytes specified by the operand into the object code. It accepts several bytes of data, which must be separated by commas and contain no spaces. The standard number format is used and arithmetic is done as usual. The "#" symbol is acceptable but ignored, as is "<". The ">" symbol may be used to specify the high byte of the label, otherwise the low byte is always taken. The ">" symbol should appear only as the first character of an expression or immediately after #. That is, the instruction DFB >LAB1-LAB2 will produce the high byte of the value of LAB1-LAB2. For example:

DFB $34,100,LAB1-LAB2,%1011,>LAB1-LAB2

is a properly formatted DFB statement which will generate the object code (hex) 34 64 DE 0B 09, assuming that LAB1=$81A2 and LAB2=$77C4.

HEX hex data

This is an alternative to DFB which allows convenient insertion of hex data. Unlike all other cases, the "$" is not required or accepted here. The operand should consist of hex numbers having two hex digits (e.g., 0F, not F). They may be separated by commas or may be adjacent. An error message will be generated if the operand contains an odd number of digits or ends in a comma or, as in all cases, contains more than 64 characters.

DS expression (Define Storage)

This does not generate any code, but simply adjusts the object code pointer so that an area equal to the value of the operand is reserved. DS -1 can be used to back up the object and address pointers one byte (for example).

KBD (KeyBoarD)

This allows a label to be equated from the keyboard during assembly. Its syntax is: LABEL KBD.

LUP expression (Loop)
—^ (end of LUP)

The syntax of this is (for example)

```
LUP 4
ASL
--^
```

This will assemble as

```
ASL
ASL
ASL
ASL
```

and will show that way in the assembly listing, with repeated line numbers.

USR (opcode)

This is a user definable pseudo opcode. It does a JSR $B6DA. (This location will contain an RTS after a boot or a BRUN MERLIN or BRUN BOOT ASM.) To set up your routine you should BRUN it from the EXECs command after CATALOG. This should just set up a JMP at $B6DA to the main routine and then RTS. The following flags and entry points may be used by your routine:

```
USRADS     = $B6DA  ;must have a JMP to your routine
PUTBYTE    = $E5F6  ;see below
EVAL       = $E5F9  ;see below
PASSNUM    = $2     ;contains assembler's pass number
ERRCNT     = $1D    ;error count
VALUE      = $55    ;value returned by EVAL
OPNDLEN    = $BB    ;contains combined length of
                      operand and comment
NOTFOUND   = $FD    ;see discussion of EVAL
WORKSP     = $280   ;contains the operand and
                      comment in positive ASCII
```

Your routine will be called by the USR opcode with A=0, Y=0 and carry set. To direct the assembler to put a byte in the object code, you should JSR PUTBYTE with the byte in A. PUTBYTE will preserve Y but will scramble A and X. It returns with the zero flag clear (so that BNE always branches). On the first pass, PUTBYTE just adjusts the object and address pointers, so that the contents of the registers are not important. You MUST call PUTBYTE the SAME NUMBER OF TIMES on each pass or the pointers will not be kept correctly and the assembly of other parts of the program will be incorrect!

If your routine needs to evaluate the operand, or part of it, you can do this by a JSR EVAL. The X register must point to the first character of the portion of the operand you wish to evaluate. (Thus, put X=0 to evaluate the expression at the start of the operand.) On return from EVAL, X will point to the character following the evaluated expression. The Y register will be 0, 1, or 2 accordingly as this character is a right parenthesis, a space or a comma.

Any other character not allowed in an expression will cause assembly to abort with a BAD OPERAND error. If some label in the expression is not recognized then location NOTFOUND will be nonzero. (On the second pass, however, you will get an UNKNOWN LABEL error and the rest of your routine will be ignored.) On return from EVAL the computed value of the expression will be in location VALUE, VALUE+1, low byte first. On the first pass this value will be insignificant if NOTFOUND is nonzero.

Appropriate locations for yur routine are $300-$3CF and $8A0-$8FF. You must not write to $900. For a longer routine, you may use high memory just below $9853. If you are not sure that the symbol table will not exceed $1000 bytes then you could use the SYM editor command to protect your routine from overwrite by the object code. (SYM would have to be set at least one byte below your code.) You may use zero page locations $60-$6F but should not alter other locations. Also,you must not change anything from $226 to $27F or anything from $2C4 to $2FF. Upon return from your routine (RTS) the USR line will be printed (on the second pass).

To gain further understanding of the use of USR, read the source file SCRAMBLE.S or, for a more sophisticated example, the file FLOAT.S. The first of these uses the USR opcode to put an ASCII string into the object code in a scrambled format. The second is a somewhat complicated routine that uses Applesoft to compute the packed (5 byte) form of a specified floating point number, and put it in the object coed. (Ther latter can be used, for assembly, only on an Apple][Plus.)

When you use the USR opcode in a source file, it is wise to include some sort of check (in source) that the required routine is in memory. If, for example, your routine contains the byte $31 at location $310 then

```
ERR (($310)-$31
```

will test that byte and abort assembly if it is not there. Similarly, if you know that the required routine should assemble exactly two bytes of data, then you can (roughly) check for it by the following code:

```
LABEL    USR  OPERAND
         ERR  *-LABEL-2
```

This will force an error on the second pass if USR does not produce exactly two object bytes.

It is possible to use USER for several different routines in the same source. For example, your routine could check the first operand expression for an index to the desired routine and act accordingly. Thus "USR 1, whatever" would branch to the first routine, "USR 2,stuff" to the second, etc.

Conditionals

DO expression

This, together with ELSE and FIN are the conditional assembly pseudo-ops. If the operand evaluates to ZERO, then the assembler will stop generating object code (until it sees another conditional). Except for macro names, it will not recognize any labels in such an area of code. If the operand evaluates to a nonzero number, then assembly will proceed as usual. This is very useful for macros. It is also useful for sources designed to generate slightly different code for different situations. For example, if you are designing a program to go to on a ROM chip, then you would want one version for the ROM and another, with small differences to create a RAM version for debugging purposes.

Similarly, in a program with text, you may wish to have one version for Apples with lower case adapters and one for those without. By using conditional assembly, modification of such programs becomes much simpler, since you do not have to make the modification in two separate versions of the source code. Every DO should be terminated somewhere later by a FIN and each FIN should be preceded by a DO. An ELSE should occur only inside such a DO, FIN structure. DO, FIN structures may be nested up to eight (8) deep (possibly with some ELSEs between). If some DO condition is off (value 0), then assembly will not resume until its corresponding FIN is encountered, or an ELSE at this level occurs. Nested DO, FIN structures are valuable for putting conditionals in MACROS.

ELSE

This inverts the assembly condition (ON-->OFF OR OFF--> ON) for the last DO.

FIN

This cancels the last DO.

Macros

MAC (MACro)

This signals the start of a MACRO definition. It must be labeled with the macro name. The name you use is then reserved and cannot be referenced by things other than the PMC pseudo-op. (Thus, things like DA NAME will not be accepted if NAME is the label on MAC. However, the samething can be simulated by preceding the macro by a LABEL EQU *, or a LABEL DS 0, etc. There is rarely any need to do this.) See the section on MACROS for details of the usage of macros.

EOM (End Of Macro)
<<< (alternative syntax)

This signals the end of the definition of a macro. It may belabeled and used for branches to the end of a macro, or one of its copies.

PMC macro name (Put MaCro)
>>> macro name (alternative syntax)

This instructs the assembler to assemble a copy of the named macro at the present location. See the section on MACROS. It may be labeled.

Variables

Labels beginning with "]" are regarded as VARIABLES. These may be defined only by EQU and cannot be used to label something else. They can be redefined as often as you please. The designed purpose of variables is for use in MACROS, but they are not confined to that use.

Forward reference to a variable is impossible (with correct results) but the assembler will assign some value to it. That is, a variable should be defined before it is used.

MACROS

Defining a macro

A macro definition begins with

NAME MAC (no operand)

with NAME in the label field. Its definition is terminated by the pseudo-op EOM or <<<. The label NAME cannot be referenced by anything other than PMC NAME (or >>> NAME).

You can simply define the macro the first time you wish to use it in the program. However, it is preferable to first define all macros at the start of the program with the assembly condition OFF and then refer to them when needed.

Forward reference to a macro definition is not possible, and would result in a NOT MACRO error message. That is, the macro must be defined before it is called by PMC.

The conditionals DO, ELSE and FIN may be used inside a macro.

Labels inside macros, such as LOOP and OUT in the example on page 5-4, are updated each time PMC is encountered. Thus they may be used to branch into the middle of a macro as long as the branch is a "backward" one.

Error messages generated by errors in macros usually abort assembly, because of possibly harmful effects. Such messages will usually indicate the line number of a PMC rather than the line inside the macro where the error occurs.

Nested Macros

Macros may be nested to a depth of 15. For nesting, macros must be defined with DO condition off.

Here is an example of a nested macro in which the definition itself is nested. (This can only be done when both definitions end at the same place.)

```
TRDB MAC
     >>> TR.]1+1;]2+1
TR   MAC
     LDA ]1
     STA ]2
     <<<
```

In this example >>> TR.LOC;DEST will assemble as:

```
     LDA LOC
     STA DEST
```

and >>> TRDB.LOC;DEST will assemble as :

```
     LDA LOC+1
     STA DEST+1
     LDA LOC
     STA DEST
```

A more common form of nesting is illustrated by these two macro definitions (where CH = $24):

```
POKE MAC
     LDA #]2
     STA ]2
     <<<

HTAB MAC
     >>> POKE.CH;]1
     <<<
```

Special Variables

Eight variables, named]1 through]8, are predefined and are designed for convenience in MACROS. These are used in a PMC statement as follows:

The instruction

```
>>> NAME expr1;expr2;expr3...
```

will assign the value of expr1 to the variable]1, that of expr2 to]2, and so on. An example of this usage is:

```
TEMP    EQU     $10
        DO      0
SWAP    MAC
        LDA     ]1
        STA     ]3
        LDA     ]2
        STA     ]1
        LDA     ]3
        STA     ]2
        <<<
        EIN
        >>>     SWAP $6;$7;TEMP
        >>>     SWAP $1000;$6;TEMP
```

(this program segment swaps the contents of location $6 with that of $7, using TEMP as a scratch depository, then swaps the contents of $6 with that of $1000.)

If, as above, some of the special variables are used in the MACRO definition, then values for them must be specified in the PMC (or >>>) statement. In the assembly listing, the special variables will be replaced by their corresponding expressions.

The assembler will accept some other characters in place of the space between the macro name and the expressions in a PMC statement. For example, you may use "/", ",","-", or "(". The semicolons are required, however, and no extra spaces are allowed.

Macros will accept literal data. Thus the assembler will accept the following type of macro call:

```
        DO   0
   MUV  MAC
        LDA  ]1
        STA  ]2
        <<<
        FIN
        >>>  MUV. (PNTR),Y;DEST
        >>>  MUV. #3;FLAG,X
```

It will also accept:

```
        DO   0
 PRINT  MAC
        JSR  SENDMSG
        ASC  ]1
        BRK
        <<<
        FIN
        >>>  PRINT.!"quote"!
        >>>  PRINT.'This is an example'
        >>>  PRINT."So's this, understand?"
```

LIMITATION: If such strings contain spaces or semicolons, they MUST be delimited by quotes (single or double). Also, literals such as >>> what."A" must have the final delimiter. (This is only true in macro calls or VAR statements, but it is good practice in all cases.)

A previous version of this assembler, that did not have this capability, used commas rather than semicolons in >>> statements. For people who have that version, a program "CONVERT" has been provided which changes these commas to semicolons in a matter of a second or two. With the source file in memory, it should be BRUN from the EXEC mode's Command after Catalog.

Sample Program

Here is a sample program intended to illustrate the usage of macros with nonstandard variable. It would, however, be simpler and more pleasing if it used]1 instead of]MSG. (In that case the variable equates should be eliminated and the values for]1 must be specified in the >>> lines.)

```
HOME      EQU     $EC58
COUT      EQU     $FDED
KEY       EQU     $C000
STROBE    EQU     $C010
DOS       EQU     $3D3
          DO      0          ;Assembly off
SENDMSG   MAC                ;Start of definition of the
                             ;macro "SENDMSG"
          LDY     #0
LOOP      LDA     ]MSG,Y     ;Get a character
          BEQ     OUT        ;End of message
          JSR     COUT       ;Send it
          INY
          BNE     LOOP       ;Back for more
OUT       <<<                ;End of macro definition and
                             ;exit from routine
          FIN                ;Turn assembly ON

          JSR     HOME       ;Clear screen
]MSG      EQU     HITMSG
          >>>     SENDMSG
INVRS     CMP     #"I"
          BNE     NORM
]MSG      EQU     IMSG
          >>>     SENDMSG
NORM      CMP     #"N"
          BNE     STP
]MSG      EQU     NMSG
          >>>     SENDMSG
STP       CMP     #"S"       ;Does he want to stop?
          BNE     GETKEY     ;No, get the next input
          JMP     DOS        ;All done, exit gracefully
HITMSG    ASC     !HIT       ;A KEY "F","I","N", OR "S"!
          HEX     8D8D00
FMSG      FLS     "THIS IS A FLASHING MESSAGE"
```

```
        HEX     8D8D00
IMSG    INV     "THIS IS A MESSAGE IN INVERSE"
        HEX     8D8D00
NMSG    ASC     "THIS IS A NORMAL MESSAGE"
        HEX     8D8D00
```

THE MACRO LIBRARY

A macro library with three example macro programs is included in source file form on this diskette. The purpose of the library is to provide some guidance to the newcomer to macros and how they can be used within an assembly program. Note all macros are defined at the beginning of the source file, then each example program places the macros where they are needed. Conditionals are used to determine which example program is to be assembled. The KBD opcode allows the user to make this selection from the keyboard during assembly.

TECHNICAL INFORMATION

SOURCE is placed at $901 when loaded, regardless of its original address.

The important pointers are:

START OF SOURCE	in	$A,$B	(always set $901)
HIMEM	in	$C,$D	(defaults to $8000)
END OF SOURCE	in	$E,$F	

When you exit to BASIC or to the monitor, these pointers are saved at $E00A-$E00F They are restored upon reentry to MERLIN.

Entry into MERLIN replaces the 1/O hooks by the standard onesand then reconnects DOS. (This is the same as typing PR#0 and IN#0 from the keyboard.) Entry to the EDITOR disconnected DOS (so that you can use labels such as INIT without disastrous consequences). Reentry to EXEC MODE disconnects any extent 1/O hooks that you may have established via the editor's PR# command, and reconnects DOS. Exit from assembly (completion of assembly or control C) also disconnects extant 1/O hooks.

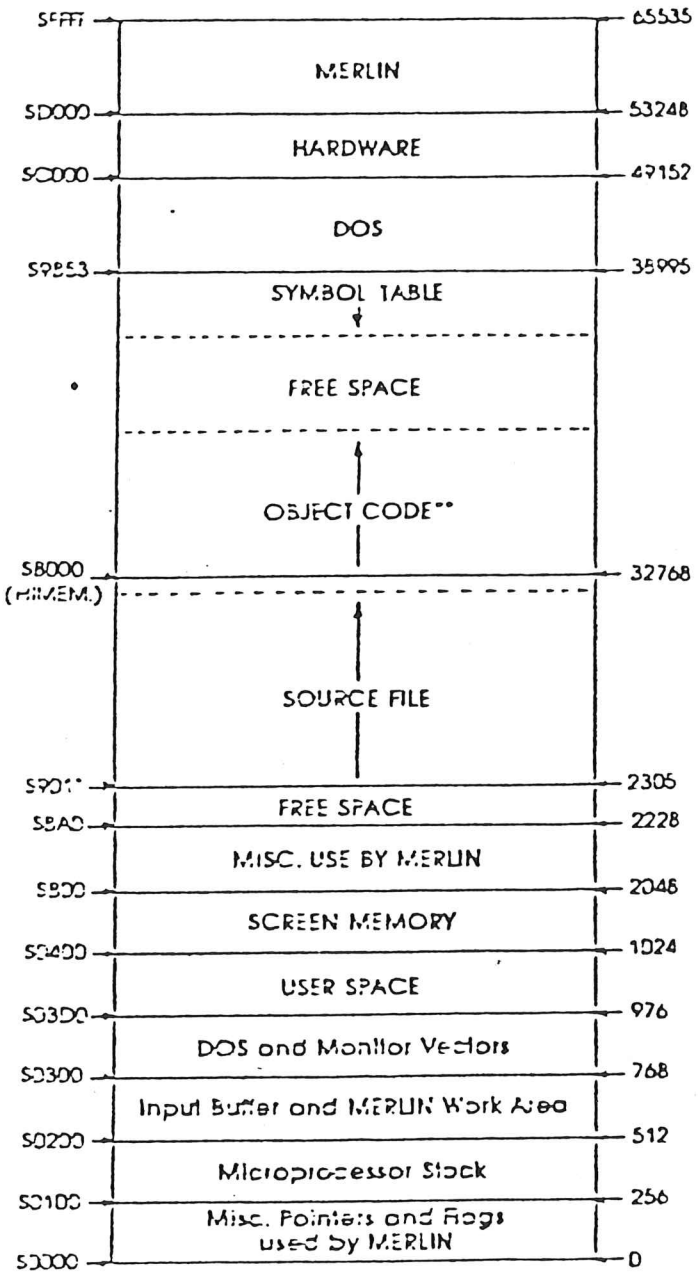
GENERAL INFORMATION

Reentry after exit to BASIC is made by the "ASSEM" command. A BRUN MERLIN or a disk boot will also provide a warm reentry and will not reload MERLIN if it is already there. This may be forced by BRUN BOOT ASM which would then be a cold entry, "destroying" any file in memory.

Memory organization, for ordinary sized files is not of concern to the user, but it is important to understand certain constraints for the handling of large files. HIMEM (which defaults to $8000) is an upper limit to the source file. It is also an upper limit for PUT files. If a memory error occurs during assembly, indicating a PUT line, it means the 5PUT file exceeded HIMEM and thus that HIMEM will have to be increased.

The default ORG and OBJ addresses equal the present value of HIMEM . It is illegal to specify an OBJ address that is less than HIMEM except that a page 3 address is allowed. (If a page 3 OBJ address is used, the user MUST be careful that the file will not write over the DOS jumps at $3D0-$3FF as the assembler does NOT check for this error.) If, during assembly, the object code exceeds BASIC HIMEM (or the SYM address, if one has been specified) then the code will not be written to memory, but assembly will appear to proceed and its output sent to the screen or printer. The only clue that this has happened, if not intentional, is that the object SAVE command is disabled in this event. Thus, if a listing for a very long file is desired, without actually creating code, the user can assemble over DOS and up.

2. MEMORY MAP



*900 also used by editor.

**OBJ and ORG default to value of HIMEM.

SYMBOL TABLE

The symbol table is printed after assembly unless LST OFF has been invoked. It comes first in aphabetical order and then in numerical order. The symbol table is flagged as follows:

MD = Macro Definition
M = Label defined in a macro (LOOP and OUT in the example)
V = Variable (symbols starting with])
? = A symbol that was never referenced

Internally, these are flagged by setting bits 7 to 4 of the symbols length byte:

?=bit 7 MD=bit 5 M=bit 4

Also, bit 6 is set during the alphabetical printout to flag printed symbols, then removed during the numerical order printout. The symbol printout is formatted for an 80 column printer, or for one which will send a carriage return after 40 columns.

USING MERLIN WITH SHIFT KEY MODIFICATIONS

MERLIN supports all hardware shift key modifications. The CONFIGURATION program will establish the modification that you want supported. MERLIN is smart enough to know if the modification actually exists in the Apple you are using and defeats the modification if it is not there. Thus it can be used on another machine without reconfiguration.

USING MERLIN WITH 80 COLUMN BOARDS

Most, but not all, 80 column boards are supported. You may use the VIDEO command to select the 80 column board. To have the board selected upon boot, use the CONFIGURATION program. (Then the VIDEO 0 or $10 command followed by RESET will switch back to the normal Apple screen.)

If your board does not support inverse, then control characters in source will show as ordinary capital letters instead of inverse letters as with boards that support inverse. You can use the editor's FIND command to search for particular control characters, however, if you wish to verify their presence or absence, or simply switch over to the normal Apple screen.

If, for example, your copy of MERLIN has not been configured to support an 80 column card in slot 3, and there is no card in that slot, then MERLIN will recognize this and defeat the 80 column provision. Thus there is no need to reconfigure for use on another computer.

MERLIN will NOT support any board that does not recognize the "POKE 36" method of tabbing. (As far as we are aware, this means simply that it will not support old versions of the FULL VIEW card.)

When in Edit mode, MERLIN takes total control of input and output. Thus the effect of typing a control character will be as described in this manual and NOT as described in the manual for your 80 column card. For example, Control-L will not blank the screen, but is the case toggle. Control-A, which acts as a case toggle on many 80 column cards, will not do this in Edit mode and it simply produces a Control-A in the file line.

THE CONFIGURE ASM PROGRAM

This program allows you to make several minor modifications to MERLIN's default conditions. It first allows you to change the "UPDATE SOURCE" character searched for at the entry to the assembler, and the editor's wild card character, and the number of symbol fields printed per line in the symbol table printout.

It also allows you to specify whether you want to have an 80 column board supported and, if so, which slot it is in.

It also allows you to specify a hardware shift key modification. Any such modification can be supported. However, if your modification is the type that enables direct input of lower case (as with the VIDEX Keyboard Enhancer) instead of providing a memory location to be tested (as with the "game button 2" modification), then the default at the start of each line will be lower case rather than upper case and Control-L will function as a case lock toggle.

You may specify whether you have a lower case adapter. This will affect the condition on boot if you have not elected to have an 80 column board selected then. (It may always be defeated from the editor using the VIDEO command, however, so this is only to select the initial condition.)

You may select a number of other options including certain printer options for use by the PRTR command.

Finally, you can save the configured version to another, or the same disk. There is no reason to keep the original version since you can always return to it by reconfiguration.

At the end of the configuration program the user is given the opportunity to transfer the boot program "MERLIN" to another disk. This is just a convenient way of transferring that program since it cannot be done by manually. You can also use FID to copy MERLIN or ASM.OBJ (the main program.)

NOTE: MERLIN will RUN only on the original disk or a protected copy. (The files may be kept for safety on another disk, but cannot function until re-copied onto a protected disk). The configuration program should be BRUN only when the standard Apple screen is in use.

ERROR MESSAGES

BAD OPCODE
Occurs when the opcode is not valid (perhaps misspelled) or the opcode is in the label column.

BAD ADDRESS MODE
The addressing mode is not a valid 6502 instruction; for example, JSR (LABEL) or LDX(LABEL),Y.

BAD BRANCH
A branch (BEQ,BCC,etc;) to an address that is out of range, i.e.; further away than ±127 bytes. Note that most errors will throw off the assembler's address calculations. Thus bad branch errors should be ignored until previous errors have been dealt with.

BAD OPERAND
An illegally formatted operand. This also occurs if you "EQU" a label to a zero page number after the label has been used. It may also mean that your operand is longer than 64 characters, or that a comment line exceeds 64 characters. This error will abort assembly.

DUPLICATE SYMBOL
On the first pass, the assembler finds two identical labels.

MEMORY FULL
On the first pass, the symbol table exceeds HIMEM ($8000 by default). This can also be caused by a PUT file exceeding HIMEM. Assembly is aborted by this error.

UNKNOWN LABEL
Your program refers to a LABEL that does not exist. This also occurs if you try to reference a MACRO definition by anything other than PMC. This can also occur if the referenced label is in an area with conditional assembly OFF. (The latter will not happen with a MACRO definition.)

NOT MACRO
Forward reference to a MACRO, or reference by PMC to a label that is not a MACRO.

NESTING ERROR
Macros nested more than 15 deep or conditionals nested more than 8 deep.

BAD "PUT"
This is caused by a PUT inside a macro or by a PUT inside another PUT file.

BAD "SAV"
This is caused by a SAV inside a macro or a SAV after a multiple OBJ after the last SAV.

BAD INPUT
This results from either no input ([RTN] alone) or an inut exceeding 37 characters in answer to the KBD opcode's request for the value of a label.

BREAK
This message is caused by the ERR opcode when the expression in the iperand is found to be nonzero.

BAD LABEL
This is caused by an unlabeled EQU or MAC, a label that is too long or one containing illegal characters.

SOURCEROR

Introduction

Sourceror is a sophisticated and easy to use disassembler designed as a subsidiary to create MERLIN source files out of binary programs, usually in a matter of minutes. Sourceror disassembles Sweet 16 code as well as 6502 code.

The main part of Sourceror is called SRCRR.OBJ, but this cannot be run (conveniently) directly, since it may overwrite DOS buffers and bomb the system. For this reason, a small program named SOURCEROR is provided. This runs in the input buffer, and thus does not conflict with any program in memory. This small program simply checks memory size, gets rid of any program such as PLE which would conflict with the main Sourceror program, sets MAXFILES 1, then runs SRCRR.OBJ (at $8800-$9AA5). To minimize the possibility of accident, SRCRR.OBJ has a default location of $4000 and if you BRUN it, it will just return without doing anything. If you try to BRUN it at its designed location of $8800, however, you could be in for big trouble. Sourceror assumes the standard Apple screen is being used and will not function with an 80 column card.

Using Sourceror

1. Load in the program to be disassembled. Although Sourceror will handle programs at any location, the original location for the program is preferable as long as it will not conflict with Sourceror and the build up of the source file. When in doubt, load it in at $800 or $803. Small programs at $4000 and above, or medium sized ones above $6000 will probably be OK at their original locations.

2. BRUN SOURCEROR

3. You will be told that the default address for the source file is $2500. This was selected because it does not conflict with the addresses of most binary programs you may wish to disassemble, Just hit RETURN to accept this default address. Otherwise, specify (in hex) the address you want.

You may also access a "secret" provision at this point. This is done by typing Control S (for "Sweet") after (or in lieu of) the source address. Then you will be asked to specify a (nonstandard) address for the Sweet 16 interpreter. This is intended to facilitate disassembly of programs which use a RAM version of Sweet 16.

4. Next, you will be asked to hit RETURN if the program to be disassembled is at its original (running) location. Otherwise, you must specify, in Hex, the present location of the code to be disassembled. Then, you will be asked to give the ORIGINAL location of that program.

When disassembling, you must use the ORIGINAL address of the program, not the address where the program currently resides. It will appear that you are disassembling the program at its original location, but actually, Sourceror is disassembling the code at its present location and translating the addresses.

5. Next, you will finally see the title page which contains a synopsis of the commands to be used in disassembly. You may now start disassembling or using any of the other commands. Your first command must include a hex address. Thereafter this is optional, as we shall explain.

At this point, and until the final processing, you may hit RESET to return to the start of the Sourceror program. If you then hit RESET once more, you will exit Sourceror and return to BASIC. (This assumes you are using the autostart monitor.)

Commands Used in Disassembly

The disassembly commands are very similar to those used by the disassembler in the Apple monitor. All commands accept a 4-digit hex address before the command letter. If this number is omitted, then the disassembly continues from its present address. A number must be specified only upon initial entry.

If you specify a number greater than the present address, a new ORG will be created.

More commonly, you will wish to specify an address less than the present default value. In this case, the disassembler checks to see if this address equals the address of one of the previous lines. If so, it simply backs up to that point. If not, then it backs up to the next used address and creates a new ORG. Subsequent source lines are "erased". It is generally best to avoid new ORGs when possible. If you get a new ORG and don't want it, try backing up a bit more until you no longer get a new ORG upon disassembly.

L (List)

This is the main disassembly command. It disassembles 20 lines of code. It may be repeated (e.g., 2000LLL will disassemble 60 lines of code starting at $2000). If a JSR to the Sweet 16 interpreter is found, disassembly is automatically switched to Sweet 16 mode.

Command L always continues the present mode (Sweet 16 or normal) of disassembly.

If an illegal opcode is encountered, then the bell will sound and opcode will be printed as three question marks in flashing format. This is only to call your attention to the situation. In the source code itself, unrecognized opcodes are converted to HEX data, but not displayed on the screen.

S (Sweet)

This is similar to L, but forces the disassembly to start in Sweet 16 mode. Sweet 16 mode returns to normal 6502 mode whenever the Sweet 16 RTN opcode is found.

N (Normal)

This is the same as L, but forces disassembly to start in normal 6502 mode.

H (Hex)

This creates the HEX data opcode. It defaults to one byte of data. If you insert a one byte (one or two digits) hex number after the H, then that number of data bytes will be generated.

T (Text)

This attempts to disassemble the data at the current address as an ASCII string. Depending on the form of the data, this will (automatically) be disassembled under the pseudo opcode ASC, DCI, INV or FLS. The appropriate delimiter " or ' is automatically chosen. The disassembly will end when the data encountered is inappropriate, when 62 characters have been treated, or when the high bit of the data changes. In the latter case the ASC opcode is automatically changed to DCI.

Sometimes the change to DCI is inappropriate. This change can be defeated by using TT instead of Tin the command.

Occasionally, the disassembled string may not stop at the appropriate place because the following code looks like ASCII data to Sourceror. In this event, you may limit the number of characters put into the string by inserting a (1 or 2 digit hex) number after the T command. This, or TT, may also have to be used to establish thecorrect boundary between a regular ASCII string and a flashing one. It is usually obvious where this should be done.

Any lower case letters appearing in the text string are shown as flashing
letters.

W (Word)

This disassembles the next two bytes at the current location as a DA opcode. Optionally, if the command WW is used, then these bytes are disassembled as a DDB opcode. Finally, if W- is used as the command, the two bytes are disassembled in the form DA LABEL-1. (The latter is often the appropriate form when the program uses the address by pushing it on the stack. You may detect this while disassembling, or only after the program has been disassembled. In the latter case, it may be to your advantage to do the disassembly again with some notes in hand.)

Housekeeping Commands

/ (Cancel)

This essentially cancels the last command. More exactly, it reestablishes the last default address (the address used for a command not necessarily attached to an address). This is a useful convenience which allows you to ignore the typing of an address when a backup is desired. For example: suppose you type T to disassemble some text. You may not know what to expect the following the text, so you can just type to L to look at it. Then if, for example, the text turns out to be followed by some hex data (such as $8D for a carriage return), then simply type / to cancel the L and type the appropriate H command.

R (Read)

This allows you to look at memory in a format that makes imbedded text stand out. To look at the data from $1234 to $1333 type 1234R. After that, R alone will bring up the next page of memory. The numbers you usefor this command are totally independant of the disassemtly address.

Thus you may disassemble, then use (address)R, then L (alone,, and the disassembly will proceed just as if you never used R at all. If you don't intend to use the default address when you return to disassembly, it may be wise to make a note on where you wanted to resume, or to use the / before the R.

Q (Quit)

This ends disassembly and goes to the final processing, which is automatic. If you type an address before the Q, then the address pointer is backed to (but not including) that point before the processing. Thus if, at the end of the disassembly, the disassembled lines include:

```
2341-  4C  03  E0      JMP  $E003
2344-  A9  BE  94      LDA  $94BE,Y
```

and the last line is just garbage, then type 2344Q. This will cancel the last line, but retain the first.

Final Processing

After the Q command, the program does some last minute processing of the assembled code. If you hit RESET at this time, you will return to BASIC and lose the disassembled code.

The processing may take from a second or twofor a short program to two or three minutes for a long one. Be patient.

When the processing is done, you are asked if you want to save the source (if you don't, it will be lost). If so, you will be asked for a file name. Sourceror will append the suffix ".S" to this name and save it to disk.

The drive used will be the one used to BRUN SOURCEROR. Thus, replace the disk first if you want the source to go on another disk.

To look at the disassembled source, BRUN MERLIN, or type ASSEM, and load it in.

Dealing with the Finished source

In most cases, after you have some experience and assuming you used reasonable care, the source will have few, if any, defects.

You may notice that some DA's would have been more appropriate in the DA LABEL-1 or the DDB LABEL formats. In this, and similar cases, it may be best to do the disassembly again with some notes in hand. The disassembly is so quick and painless, this is often much easier than trying to alter the source appropriately.

The source will have all the exterior or otherwise unrecognized labels at the end in a table of equates. You should look at this table closely. It should not contain any zero page equates except ones resulting from DA's or JMP's or JSR's. This is almost a sure sign of an error (yours, not Sourceror's) in the disassembly. It may have resulted from an attempt to disassemble a data area as regular code. Note that if you try to assemble the source under these conditions, you will get an error as soon as the equates appear. If, as eventually you should, you move the equates to the start of the program. then you will not get an error, but the assembly MAY NOT BE CORRECT. Thus, it is important to deal with this situation first. The trouble here is for the following reason. If for example, the disassembler finds the data AD 00 8D, it will disassemble it, correctly, as LDA $008D. The assembler, however always assembles this code as a zero page instruction, giving the two bytes A5 8D. Occasionally you will find a program that uses this form for a zero page instruction. In that case, you will have to insert a character after the LDA opcode to have it assemble identically to its original form. More usually, however,it was data in the first place, rather than code, and must be dealt with to get a correct assembly.

The Memory Full Message

When the source file reaches within $600 of the start of Sourceror (that is, when it goes beyond $8200) you will see "MEMORY FULL" and "HIT A KEY" in flashing format. When you hit a key, Sourceror will go directly to the final processing. The reason for the $600 gap is that Sourceror needs a certain amount of space for this processing. It is possible (but not likely) that part of Sourceror will be overwritten during final processing, but this should not cause problems since the front end of Sourceror will not be used again at that point. There is a "secret" override provision at the memory full point. If the key you hit is Control O (for override), then Sourceror will return for another command. You can use this to specify the desired ending point. You can also use it to go a little farther than Sourceror wants you to, and disassemble a few more lines. Obviously, you should not carry this to extremes.

CAUTION: After exiting Sourceror, do not try to run it again with a CALL. Instead, run it again from disk. This is because the DOS buffers have been reestablished upon exit, and have partially destroyed sourceror.

The LABELER program

One of the nicest features of the SOURCEROR program is the automatic assignment of labels to all recognizable addresses in the binary file being disassembled. Addresses are recognized by being found in a table which SOURCEROR references during the disassembly process. For example, all JSR $FC58 instructions within a binary file will be listed by SOURCEROR as JSR HOME. This table of address labels may be edited by using the program LABELER.

To use labeler, BRUN LABELER. The program will then mention that SRCRR.OBJ is being loaded into memory, and then present the main program menu.

The commands are as follows:

Q:QUIT

When finished with any modifications you wish to make to the label table, press 'Q' to exit the LABELER program. If you wish to save the new file, press 'S'. Otherwise, press ESCAPE to exit without saving the table, if for instance you had just been reviewing the table.

L:LIST

This allows you to list the current label table. After 'L', press any key to start the listing. Pressing any key will go to the next page; Control-C will abort the listing.

D:DELETE LABEL(S)

Use this option to delete any address labels you do not want in the list. After entering the D command, simply enter the NUMBER of the label you want to delete. If you want to delete a range, enter the beginning and ending label numbers, separated by a comma.

A:ADD LABEL

Use this option to add a new label to the list. Simply tell the program the hex address and the name you wish to associated with that address. Press RETURN only, to abort this option at any point.

P:FREE SPACE

This tells you how much free space remains in the table for new label entries.

U:UNLOCK SRCRR.OBJ

Before saving a new label table, you will need to unlock the SRCRR.OBJ file. Use this command before Quitting the LABELER program, if you intend to save a new file.

SWEET 16

by Dick Sedgewick

Introduction

Sweet 16 is probably the least used and least understood seed in the Apple][.

In exactly the same sense that Integer and Applesoft Basics are languages, Sweet 16 is a language. Compared to the Basics, however, it would be classed as a low level language, with a strong likeness to conventional 6502 Assembly language.

To use Sweet 16, you must learn the language - and to quote "WOZ", "The op code list is short and uncomplicated". "WOZ", of course is Mr. Apple, and the creator of Sweet 16.

Sweet 16 is ROM based in every Apple][from $F689 to $F7FC. It has its own set of op codes and instruction sets, and uses the SAVE and RESTORE routines from the Apple Monitor to preserve the 6502 registers when in use, allowing Sweet 16 to be used as a subroutine.

It uses the first 32 locations on zero page to set up its 16 double byte registers, and is therefore not compatible with Applesoft Basic without some additional efforts.

Theoriginal article, "Sweet 16: The 6502 Dream Machine", first appeared in Byte Magazine, November 1977 and later in the original "WOZ PAK". This article is included here and again as test material to help understand the use and implementation of Sweet 16.

Examples of the use of Sweet 16 are found in the Programmer's Aid #1, in the Renumber, Append, and Relocate programs. The Programmers Aid Operating Manual contains complete source assembly listings, indexed on page 65.

The demonstration program is written to be introductory and simple, consisting of three parts:

1. Integer Basic Program
2. Machine Language Subroutine
3. Sweet 16 Subroutine

The task of the program will be to move data. Parameters of the move will be entered in the Integer Basic Program.

The "CALL 768" ($300) at line 120, enters a 6502 machine language subroutine having the single purpose of entering a Sweet 16 and subsequently returning to BASIC (addresses $300, $301, $302, and $312 respectively). The Sweet 16 subroutine of course performs the move, and is entered at hex locations $303 to $311 (see listing Number 3).

After the move, the screen will display three lines of data, each 8 bytes long, and await entry of a new set of parameters. The three lines of data displayed on the screen are as follows:

Line 1: The first 8 bytes of data starting at $800, which is the fixed source of data to be moved (in this case, the string A$).

Line 2: The first 8 bytes of data starting at the hex address entered as the destination of the move (high order byte only).

Line 3: The first 8 bytes of data starting at $0000 (the first four Sweet 16 registers).

The display of 8 bytes of data was chosen to simplify the illustration of what goes on.

Integer Basic has its own way of recording the string A$. Because the name chosen for the string "A$" is stored in 2 bytes, a total of five housekeeping bytes precede the data entered as A$, leaving only three additonal bytes available for display. Integer Basic also adds a housekeeping byte at the end of a string , known as the "string terminator".

Consequently, for the purposes of the convenience of the display, and to see the string terminator as the 8th byte, the string data entered via the keyboard should be limited to two characters, and will appear as the 6th and 7th bytes. Additionally, parameters to be entered include the number of bytes to be moved. A useful range for this demonstration would be 1-8 inclusive, but of course 1-255 will work.

Finally, the starting address of the destination of the move must be entered. Again, for simplicity, the high order byte only is entered, and the program allows a choice between Decimal 9 and H.O.B. of program pointer 1, to avoid unnecessary problems (in this demonstration enter a decimal number between dec 9 and 144 for a 48K APPLE).

The8 bytes of data displayed starting at $00 will enable one to observe the condition of the Sweet 16 registers after a move hasbeen accomplished, and thereby understand how the Sweet 16 program works.

From the article "Sweet 16: The 6502 Dream Machine", it will be remembered that Sweet 16 can establish 16 double byte registers, starting at $00. That means that Sweet 16 can use the first 32 addresses on zero page.

The "events" occuring in this demonstration program can be studied in the first four Sweet 16 registers, therefore the 8 byte display starting at $0000 is large enough for this purpose.

These four registers are established as R0, R1, R2, R3:

R0	$0000	&	0001	-Sweet 16 accumulator
R1	$0002	&	0003	-Source address
R2	$0003	&	0004	-Destination address
R3	$0004	&	0005	-Number of bytes to move
.				
.				
.				
R14	$001C	&	001D	-Prior result register
R15	$001E	&	001F	-Sweet 16 Program counter

Additionally, an examination of registers R14 and R15 will extend an understanding of Sweet 16, being fully explained in the "WOZ" text. Notice that the HOB of R14, (located at $1D) contains $06, which is the doubled register specification (3X2=$06). R15, theSweet 16 program counter contains the address of the next operation, (as it did for each step during execution of the program), which was $0312 when execution ended, and 6502 machine code resumes.

To try a sample run, enter the Integer basic program as shown in Listing #1. Of course, REM statements can be omitted, and line 10 is only helpful if the machine code is to be stored on disk.

The Listing #2 must also be entered starting at $300. Note that a 6502 disassembly does not look like Listing #3, but the included Sourcerer disassembler would create a correct dissassembly.

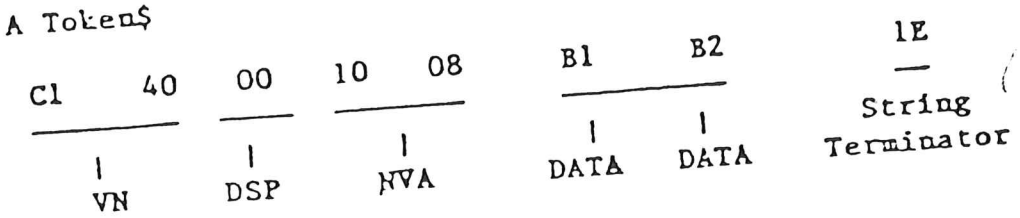
Enter RUN and RETURN
Enter 12 and hit RETURN (A$ - A$ string data)
Enter 18 and hit RETURN(High order byte of destination)

the display should appear as follows:

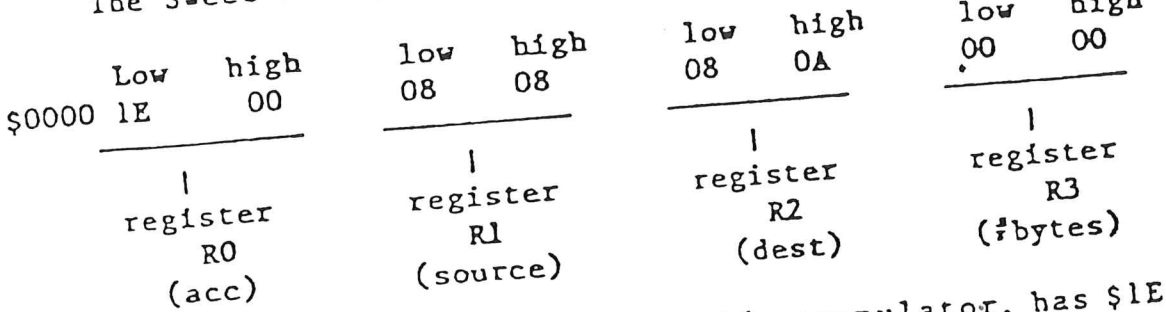
$0800-C1 40 00 10 08 B1 B2 1E (SOURCE)
$0A00-C1 40 00 10 08 B1 B2 1E (Dest.)
$0000-1E 00 08 08 08 0A 00 00 (Sweet 16)

Note that the 8 bytes stored at $0A00 are identical to the 8 bytes starting at $0800, indicating an accurate move of 8 bytes length as been made. It will be seen that they are moved one byte at a time starting with token C1 and ending with token 1E (if moving less that 8 bytes, the data following the moved data would be whatever exists at those locations before the move).

The bytes have the following significance:



The Sweet 16 registers are shown:



The low order byte of R0, the Sweet 16 accumulator, has $1E in it, which ws the last byte moved, (the 8th).

The low order byte of the source register R1 started as $00 and was incremented eight times, once for each byte of moved data.

The high order byte of the destination register R2 contains $0A, which was entered as 10 (the variable!) and poked into the Sweet 16 code. The LOB of R2 was incremented exactly like R1.

Finally, register R3, the register that stores the number of bytes to be moved has been poked to 8 (the variable B) and decremented eight times as each byte got moved, thereby ending up $0000.

By entering character strings and varying the number of bytes to be moved, the Sweet 16 registers can be observed and in fact, the contents predicted.

Working with this demonstration program, and study of the text material will soon enable the writing of Sweet 16 programs to perform additional 16 bit manipulations. The unassigned op codes mentioned in the WOZ "Dream Machine" article should present a most interesting opportunity to "play".

Sweet 16 as a language - or tool - opens a new direction to Apple][owners without spending a dime, and it's been there all the time.

For "Appleites" who desire to learn machine language programming, Sweet 16 can be used as a starting point. Having less op codes to learn and excellent text material to use, one could be effective very soon.

For those without Integer Basic, Sweet 16 is supplied as a source file on this diskette.

LISTING #1

```
>List
   10     PRINT "[D]BLOAD SWEET":  REM CTRL D
   20     CALL - 936:  DIM A $ (10)
   30     INPUT  "ENTER STRING A $ " , A $
   40     INPUT  "ENTER # BYTES " , B
   50     IF NOT B THEN 40 : REM AT LEAST 1
   60     POKE  778  B : REM POKE LENGTH
   70     INPUT  "ENTER DESTINATION" , A
   80     IF A > PEEK (203) - 1 THEN 70
   90     IF A < PEEK (205) + 1 THEN 70
  100     POKE  776 , A : REM POKE DESTINATION
  110     M = 8 :  GOSUB 160 :  REM DISPLAY
  120     CALL  768 : REM GOTO $0300
  130     M = A :  GOSUB 160 :  REM DISPLAY
  140     M = 0 :  GOSUB 160 :  REM DISPLAY
  150     PRINT :  PRINT :  GOTO 30
  160     POKE 60 , 0 :  POKE  61 , M
  170     CALL  -605 :  RETURN  :  REM  XAMB  IN MONITOR
```

LISTING #2

Enter code as follows:

```
                                                A       B
300:20  89 F6 11  00  08  12  00  00  13  00  00  41  51  52
    F3  07 FB 00  60
```

LISTING #3

```
Sweet 16
$300  20  89  F6   JSR   $F689
$303  11  00  08   SET   R1  source address
$306  12  00  00   SET   R2  destination address
                   A
$309  13  00  00   SET   R3  length
                   B
$30C  41           LD    @R1
$30D  52           ST    @R2
$30E  F2           DCR   R3
$30F  07           BNZ   $30C
$311  00           RTN
$312  60           RTS
```

Data will be poked from the Integer Basic program -

"A" from Line 100
"B" from Line 60

Sweet 16: A Pseudo 16 Bit Microprocessor

By Steve Wozniak

DESCRIPTION

While writing APPLE BASIC for a 6502 microprocessor, I repeatedly encountered a variant of MURPHY''S LAW. Briefly stated, any routine operating on 16-bit data will require at least twice the code that it should. Programs making extensive use of 16-bit pointers (such as compilers, editors, and assemblers) are included in this category. In my case, even the addition of a few double-byte instructions to the 6502 would have only slightly alleviated the problem. What I really needed was a 6502/RCA 1800 hybrid - an abundance of 16-bit registers and excellent pointer capability. My solution was to implement a non-existent (meta) 16-bit processor in software, interpreter style, which I call SWEET 16.

SWEET 16 is based on sixteen 16-bit registers (R0-15), actually 32 memory locations. R0 doubles as the SWEET 16 accumulator (ACC), R15 as the program counter (PC), and R14 as the status register. R13 holds compare instruction results and R12 is the subroutine return stack pointer if SWEET 16 subroutines are used. All other SWEET 16 registers are at the user's unrestricted disposal.

SWEET 16 instructions fall into register and non-register categories. The register ops specify one of the sixteen registers to be used as either a data element or a pointer to data element or a pointer to data in memory, depending on the specific instruction. For example INR R5 uses R5 as data and ST @R7 uses R7 as a pointer to data in memory. Except for the SET instruction, register ops take 1 byte of code each. The non-register ops are primarily 6502 style branches with the second byte specifying a +-127 byte displacement relative to the address of the following instruction. Providing that the prior register op result meets a specified branch condition, the displacement is added to SWEET 16 PC, effecting a branch.

SWEET 16 is intended as a 6502 enhancement package, not a stand-alone processor. A 6502 program switches to SWEET 16 mode with a subroutine call and subsequent code is interpreted as SWEET 16 instructions. The nonregister op RTN returns the user program to 6502 mode after restoring the internal register contents (A, S, Y, P, and S). The following example illustrates how to use SWEET 16.

```
300  B9  00  02        LDA   IN,Y      get a char.
303  C9  CD            CMP   #"M"      "M" for move
305  D0  09            BNE   NOMOVE    No. skip move
307  20  89  F6        JSR   SW16      Yes, call SWEET 16
30A  41          MLOOP LD    @R1       R1 holds source addr.
30B  52                ST    @R2       R2 holds dest. addr.
30C  F3                DCR   R3        Decr. length
30D  07  FB            BNZ   MLOOP     Loop until done
30F  00                RTN             Return to 6502 mode.
310  C9  C5       NOMOVE CMP   #"E"     "E" char?
312  D0  13            BEQ   EXIT       Yes, exit
314  C8                INY              No, cont.
```

Note: Registers A, X, Y, P, and S are not disturbed by SWEET 16.

INSTRUCTION DESCRIPTIONS

The SWEET 16 opcode listing is short and umcomplicated. Excepting relative branch displacements, hand asssembly is trivial. All register opcodes are formed by combining two hex digits, one for the opcode and one to specify a register. For example, opcodes 15 and 45 both specify register R5 while codes 23, 27 and 29 are all ST ops. Most register ops are assigned in complementary pairs to facilitate remembering them. Thus LD and ST are opcodes 2N and 3N respectively, while LD @ and ST @ are codes 4N and 5N.

Opcodes 0 to C (hex) are assigned to the thirteen non-register ops. Except for RTN (opcode 0), BK (0A), and RS (0B), the non register ops are 6502 style branches. The second byte of a branch instruction contains a +/-127 byte displacement value (in two's complement form) relative to the address of the instruction immediately following the branch.

If a specified branch condition is met by the prior register op result, the displacement is added to the PC effecting a branch. Except for BR (Branch always) and BS (Branch to Subroutine), the branch opcodes are assigned in complementary pairs, rendering them easily remembered for hand coding. For example, Branch if Plus and Branch if Minus are opcodes 4 and 5 while Branch if Zero and Branch if NonZero are opcodes 6 and 7.

SWEET 16 OPCODE SUMMARY

REGISTER OPS

1n	SET	Rn	Constant (Set)
2n	LD	Rn	(Load)
3n	ST	Rn	(Store)
4n	LD	@Rn	(Load Indirect)
5n	ST	@Rn	(Store Indirect)
6n	LDD	@Rn	(Load Double Indirect)
7n	STD	@Rn	(Store Double Indirect)
8n	POP	@Rn	(Pop Indirect)
9n	STP	@Rn	(Store POP Indirect)
An	ADD	Rn	(Add)
Bn	SUB	Rn	(Sub)
Cn	POPD	@Rn	(Pop Double Indirect)
Dn	CPR	Rn	(Compare)
En	INR	Rn	(Increment)
Fn	DCR	Rn	(Decrement)

NON-REGISTER OPS

00	RTN		(Return to 6502 mode)
01	BR	ea	(Branch always)
02	BNC	ea	(Branch if No Carry)
03	BC	ea	(Branch if Carry)
04	BP	ea	(Branch if Plus)
05	BM	ea	(Branch if Minus)
06	BZ	ea	(Branch if Zero)
07	BNZ	ea	(Branch if NonZero)
08	BM1	ea	(Branch if Minus 1)
09	BNM1	ea	(Branch if Not Minus 1)
0A	BK		(Break)
0B	RS		(Return from Subroutine)

0C	BS	ea	(Branch to Subroutine)
0D			(Unassigned)
0E			(Unassigned)
0F			(Unassigned)

REGISTER INSTRUCTIONS

SET Rn,Constant 1n Low High (Set)

The 2-byte constant is loaded into Rn (n=0 to F, hex) and branch conditions set accordingly. The carry is cleared.

EXAMPLE:

```
15  34  A0  SET   R5, $A034    R5 now contains $A034
```

LDRn 2n (Load)

The ACC (R0) is loaded from Rn and branch conditions set according to the data transferred. The carry is cleared and contents of Rn are not disturbed.

EXAMPLE:

```
15  34  A0  SET   R5, $A034
25          LD    R5          ACC now contains $A034
```

ST Rn 3n (Store)

The ACC is stored into Rn and branch conditions set according to the data transferred. The carry is cleared and the ACC contents are not disturbed.

EXAMPLE:

```
25  LD  R5
36  ST  R6      Copy the contents
                of R5 to R6
```

LD @Rn 4n (Load Indirect)

The low-order ACC byte is loaded from the memory location whose address resides in Rn and the high-order ACC byte is cleared. Branch conditions reflect the final ACC contents which will always be positive and never minus 1. The carry is cleared. After the transfer, Rn is incremented by 1.

EXAMPLE:

```
15  34  A0  SET   R5 , $A034
45          LD    @R5          ACC is loaded from memory
                                location $A034
                                R5 is incr to $A035
```

ST @Rn 5n (Store Indirect)

The low-order ACC byte is stored into the memory location whose address resides in Rn. Branch conditions reflect the 2-byte ACC contents. The carry is cleared. After the transfer Rn is incremented by 1.

EXAMPLE:

```
15  34  A0  SET   R5, $A034    Load pointers R5, R6 with
16  22  90  SET   R6, $9022    $A034 and $9022
45          LD    @R5          Move byte from $A034 to $9022
56          ST    @R6          Both ptrs are incremented
```

LDD @Rn 6n (Load double-byte Indirect)

The low order ACC byte is loaded from memory location whose address resides in Rn, and Rn is then incremented by 1. The high order ACC byte is loaded from the memory location whose address resides in the incremented Rn, and Rn is again incremented by 1. Branch conditions reflect the final ACC contents. The carry is cleared.

EXAMPLE:

```
15  34  A0  SET   R5 , $A034   The low-order ACC byte is load
65          LDD   @R6          ed from $A034, high-order from
                               $A035, R5 is incr to $A036
```

to STD @Rn 7n (Store double-byte Indirect)

The low-order ACC byte is stored into memory location whose address resides in Rn, and Rn is then incremented by 1. The high-order ACC byte is stored into the memory location whose address resides in the incremented Rn, and Rn is again incremented by 1. Branch conditions reflect the ACC contents which are not disturbed. The carry is cleared.

EXAMPLE:

```
15  34  A0  SET   R5, $A034    Load pointers R5, R6
16  22  90  SET   R6, $9022    with $A034 AND $9022
65          LDD   @R5          Move double byte from
76          STD   @R           $A034-35 TO $9022-23.
                               Both pointers incremented
                               by 2.
```

POP @Rn 8n (Pop Indirect)

The low-order ACC byte is loaded from the memory location whose address resides in Rn after Rn is decremented by 1, and the high order ACC byte is cleared. Branch conditions reflect the final 2-byte ACC contents which will always be positive and never minus 1. The carry is cleared. Because Rn is decremented prior to loading the ACC, single byte stacks may be implemented with the ST @Rn and POP @Rn ops (Rn is the stack pointer).

EXAMPLE:

```
15  34  A0  SET   R5, $A034    Init stack pointer
10  04  00  SET   R0, 4        Load 4 into ACC
55          ST    @R5          PUSH 4 onto stack
10  05  00  SET   R0, 5        Load 5 into ACC
55          ST    @R5          Push 5 onto stack
10  06  00  SET   R0, 6        Load 6 into ACC
55          ST    @R5          Push 6 onto stack
85          POP   @R5          Pop 6 off stack into ACC
85          POP   @R5          Pop 5 off stack
85          POP   @R5          Pop 4 off stack
```

STP @Rn 9n (STORE POP Indirect)

The low-order ACC byte is stored into the memory location whose address resides in Rn after Rn is decremented by 1. Branch conditions will reflect the 2-byte ACC contents which are not modified. STP @Rn and POP @Rn are used together to move data blocks beginning at the greatest address and working down. Additionally, single-byte stacks may be implemented with the STP @Rn ops.

EXAMPLE:

```
14  34  A0   SET   R4, $A034    Init pointers
15  22  90   SET   R5, $9022
84           POP   @R4          Move byte from
95           STP   @R5          $A033 to $9021
84           POP   @R4          Move byte from
95           STP   @R5          $A032 to $9020
```

ADD Rn An (Add)

The contents of Rn are added to the contents of ACC (R) and the low-order 16 bits of the sum restored in ACC. The 17th sum bit becomes the carry and the other branch conditions reflect the final ACC contents.

EXAMPLE:

```
10  34  76   SET   R0, $7634    Init R0 (ACC) and R1
11  27  42   SET   R1, $4227
A1           ADD   R1           Add R1 (sum =B85B c clear)
A0           ADD   R0           Double ACC (R0) to $70B6
                                with carry set.
```

SUB Rn Bn (Subtract)

The contents of Rn are Subtracted from the ACC contents by performing a two's complement addition:

$$ACC = ACC + \overline{Rn} + 1$$

The low-order 16 bits of the subtraction are restored in the ACC, the 17th sum bit becomes the carry and other branch conditions reflect the final ACC contents. If the 16-bit unsigned ACC contents are greater than or equal to the 16-bit unsigned Rn contents, then the carry is set, otherwise it is cleared. Rn is not disturbed.

EXAMPLE:

```
10  34  76   SET   R0, $7634    Init R0 (ACC)
11  27  42   SET   R1, $4227    and R1
B1           SUB   R1           Subtract R1
                                (diff=$340D with c set)
B0           SUB   R0           Clears ACC. (R0)
```

POP @Rn Cn (Pop Double-byte Indirect)

Rn is decremented by 1 and the high-order ACC byte is loaded from the memory location whose address now resides in Rn. Rn is again decremented by 1 and the low-order ACC byte is loaded from the corresponding memory location. Branch conditions reflect the final ACC contents. The carry is cleared. Because Rn is decremented prior to loading each of the ACC halves, double-byte stacks may be implemented with the STD @Rn and POPD @Rn ops (Rn is the stack pointer).

EXAMPLE:

```
15  34  A0   SET   R5, $A034    Init stack pointer
10  12  AA   SET   R0, $AA12    Load $AA12 into ACC.
75           STD   @R5           Push $AA12 onto stack
10  34  BB   SET   R0, $BB34    Load $BB34 into ACC.
75           STD   @R5           Push $BB34 onto stack
C5           POPD  @R5           Pop $BB34 off stack
C5           POPD  @R5           Pop $AA12 off stack
```

CPR Rn Dn (Compare)

The ACC (R0) contents are compared to Rn by performing the 16-bit binary subtraction ACC-Rn and storing the low order 16 difference bits in R13 for subsequent branch tests. If the 16-bit unsigned ACC contents are greater than or equal to the 16-bit unsigned Rn contents, then the carry is set, otherwise it is cleared. No other registers, including ACC and Rn are disturbed.

EXAMPLE:

```
15  34  A0        SET   R5, $A034    Pointer to memory
16  BF  A0        SET   R6, $A0BF    Limit address
B0          LOOP1 SUB   R0           Zero data
75                STD   @R5          Clear 2 locns
                                     Increment R5 by 2
25                LD    R5           Compare pointer R5
D6                CPR   R6           to limit R6
02  FA            BNC   LOOP1        Loop if c clear
```

INR Rn En (Increment)

The contents of Rn are incremented by 1. The carry is cleared and other branch conditions reflect the incremented value.

EXAMPLE:

```
15  34  A0  SET   R5, $A034   (Pointer)
B0          SUB   R0          Zero to R0
55          ST    @R5         Clr Locn $A034
E5          INR   R5          Incr R5 to $A036
55          ST    @R5         Clrs locn $A036
                              (not $A035)
```

DCR Rn Fn (Decrement)

The contents of Rn are decremented by 1. The carry is cleared and other branch conditions reflect the decremented value.

EXAMPLE (Clear 9 bytes beginning at location A034)

```
15  34  A0          SET   R5, $A034   Init Pointer
14  09  00          SET   R4, 9       Init counter
B0                  SUB   R0          Zero ACC
55          LOOP2   ST    @R5         Clear a mem byte
F4                  DCR   R4          Decrement count
07  FC      .       BNZ   LOOP2       Loop until Zero
```

NON-REGISTER INSTRUCTIONS

RTN 00 (Return to 6502 mode)

Control is returned to the 6502 and program execution continues at the location immediately following the RTN instruction. The 6502 registers and status conditions are restored to their original contents (prior to entering Sweet 16 mode).

BR ea 01 d (Branch always)

An effective address (ea) is calculated by adding the signed displacement byte (d) to the PC. The PC contains the address of the instruction immediately following the BR, or the address of the BR op plus 2. The displacement is a signed two's complement value from -128 to +127. Branch conditions are not changed.

Note that the effective address calculation is identical to that for 6502 relative branches. The hex add & subtract features of the APPLE][monitor may be used to calculate displacements.

```
d = $80     ea = PC + 2 - 128
d = $81     ea = PC + 2 - 127

d = $FF     ea = PC + 2 - 1
d = $00     ea = PC + 2 + 0
d = $01     ea = PC + 2 + 1

d = $7E     ea = PC + 2 + 126
d = $7F     ea = PC + 2 + 127
```

EXAMPLE:

```
$300:  01  50     BR  $352
```

BNC ea 02 d (Branch if No Carry)

A branch to the effective address is taken only if the carry is clear, otherwise execution resumes as normal with the next instruction. Branch conditions are not changed.

BC ea 03 d (Branch if Carry set)

A branch is effected only if the carry is set. Branch conditions are not changed.

BP ea 04 d (Branch if Plus)

A branch is effected only if the prior 'result' (or most recently transferred data) was positive. Branch conditions are not changed.

EXAMPLE: (Clear mem from A034 to A03F)

```
15  34  A0          SET   R5, $A034    Init pointer
14  3F  A0          SET   R4, $A03F    Init limit
B0          LOOP 3  SUB   R0
55                  ST    @R5          Clear mem byte
                                       ;Increment R5
24                  LD    R4           Compare limit
D5                  CPR   R5           to pointer
04  FA              BP    LOOP3        Loop until done
```

BM ea 05 d (Branch if Minus)

A branch is effected only if prior 'result' was minus (negative, MSB = 1). Branch conditions are not changed.

BZ ea 06 d (Branch if Zero)

A branch is effected only if the prior 'result' was zero. Branch conditions are not changed.

BNZ ea 07 d (Branch if NonZero)

A branch is effected only if the prior 'result' was non-zero. Branch conditions are not changed.

BM1 ea 08 d (Branch if Minus 1)

A branch is effected only if the prior 'result' was minus 1 ($FFFF hex). Branch conditions are not changed.

BK 0A (Break)

A 6502 BRK (break) instruction is executed. Sweet 16 may be reentered non destructively at SW16d after correcting the stack pointer to its value prior to executing the BRK.

RS 0B (Return from Sweet 16 subroutine)

RS terminates execution of a Sweet 16 subroutine and returns to the Sweet 16 calling program which resumes execution (in Sweet 16 mode). R12, which is the Sweet 16 subroutine return stack pointer, is decremented twice. Branch conditions are not changed.

BS ea 0C d (Branch to Sweet 16 subroutine)

A Branch to the effective address (PC + 2 + d) is taken and execution is resumed in Sweet 16 mode. The current PC is pushed onto a 'Sweet 16 subroutine return address' stack whose pointer is R12, and R12 is incremented by 2. The carry is cleared and branch conditions set to indicated the current ACC contents.

EXAMPLE: (Calling a 'memory move' subroutine to move A034-A03B to 3000-3007)

```
15  34  A0          SET   R5, $A034    Init pointer 1
14  3B  A0          SET   R4, $A03B    Init limit 1
16  00  30          SET   R6, $3000    Init pointer 2
0C  15              BS    MOVE         Call move subrtn

45          MOVE    LD    @R5          Move one
56                  ST    @R6          byte
24                  LD    R4
D5                  CPR   R5           Test if done
04  FA              BP    MOVE
0B                  RS                 ;Return
```

THEORY OF OPERATION

Sweet 16 execution mode begins with a subroutine call to SW16. All 6502 registers are saved at this time, to be restored when a Sweet 16 RTN instruction returns control to the 6502. If you can tolerate indefinite 6502 register contents upon exit, approximately 30 usec may be saved by entering a SW16 + 3. Because this might cause an inadvertent switch from hex to decimal mode, it is advisable to enter at SW16 the first time through.

After saving the 6502 registers, Sweet 16 initializes its PC (R15) with the subroutine return address off the 6502 stack. Sweet 16's PC points to the location preceding the next instruction to be executed. Following the subroutine call are 1-,2-, and 3-byte Sweet 16 instructions, stored in ascending memory locations like 6502 instructions. The main loop at SW16B repeatedly calls the 'execute instruction' routine to execute it.

Subroutine SW16C increments the PC (R15) and fetches the next opcode, which is either a register op of the form OP REG with OP between 1 and 15 or a non-register op of the form 0 OP with OP between 0 and 13. Assuming a register op, the register specification is doubled to account for the 3 byte Sweet 16 registers and placed in the X-reg for indexing. Then the instruction type is determined. Register ops place the doubled register specification in the high order byte of R 14 indicating the 'prior result register' to subsequent branch instructions. Non register ops treat the register specification (right-hand half-byte) as their opcode, increment the Sweet 16 PC to point at the displacement byte of branch instructions, load the A-reg with the 'prior result register' index for branch condition testing, and clear the Y-reg.

WHEN IS A RTS REALLY A JSR?

Each instruction type has a corresponding subroutine. The subroutine entry points are stored in a table which is directly indexed into by the opcode. By assigning all the entries to a common page, only a single byte of address need be stored per routine. The 6502 indirect jump might have been used as follows to transfer control to the appropriate subroutine.

```
LDA     #ADRH       High-order byte.
STA     IND+1
LDA     OPTBL,X     Low-order byte.
STA     IND
JMP     (IND)
```

To save code, the subroutine entry address (minus 1) is pushed onto the stack, high-order byte first. A 6502 RTS (return from subroutine) is used to pop the address off the stack and into the 6502 PC (after incrementing by 1). The net result is that the desired subroutine is reached by executing a subroutine return instruction!

OPCODE SUBROUTINES

The register op routines make use of the 6502 'zero page indexed by X' and 'indexed by X indirect' addressing modes to access the specified registers and indirect data. The 'result' of most register ops is left in the specified register and can be sensed by subsequent branch instructions, since the register specification is saved in the high-order byte of R14. This specification is changed to indicate R0 (ACC) for ADD and SUB instructions and R13 for the CPR (compare) instruction.

Normally the high-order R14 byte holds the 'prior result register' index times 2 to account for the 2-byte Sweet 16 registers and thus the LSB is zero. If ADD, SUB, or CPR instructions generate carries, then this index is incremented, setting the LSB.

The SET instruction increments the PC twice, picking up data bytes in the specified register. In accordance with 6502 convention, the low-order data byte precedes the high-order byte.

Most Sweet 16 nonregister ops are relative branches. The corresponding subroutines determine whether or not the 'prior result' meets the specified branch condition and if so, update the Sweet 16 PC by adding the displacement value (-128 to +127 bytes).

The RTN op restores the 6502 register contents, pops the subroutine return stack and jumps indirect through the Sweet 16 PC. This transfers control to the 6502 at the instruction immediatey following the RTN instruction.

The BK op actually executes a 6502 break instruction (BRK), transferring control to the interrupt handler.

Any number of subroutine levels may be implemented within Sweet 16 code via the BS (Branch to Subroutine) and RS (Return from Subroutine) instructions. The user must initialize and otherwise not disturb R12 if the Sweet 16 subroutine capability is used since it is utilized as the automatic subroutine return stack pointer.

MEMORY ALLOCATION

The only storage that must be allocated for Sweet 16 variables are 32 consecutive locations in page zero for the Sweet 16 registers, four locations to save the 6502 register contents, and a few levels of the 6502 subroutine return address stack. If you don't need to preserve the 6502 register contents, delete the SAVE and RESTORE subroutines and the corresponding subroutine calls. This will free the four page zero locations ASAV, XSAV, YSAV, and PSAV.

USER MODIFICATIONS

You may wish to add some of your own instruction to this implementation of Sweet 16. If you use the unassigned opcodes $0E and $0F, remember that Sweet 16 treats these as 2-byte instructions. You may wish to handle the break instruction as a Sweet 16 call, saving two bytes of code each time you transfer into Sweet 16 mode. Or you may wish to use the Sweet 16 BK (break) op as a 'CHAROUT' call in the interrupt handler. You can perform absolute jumps within Sweet 16 by loading the ACC (R0) with the address you wish to jump to (minus 1) and executing a ST R15 instruction.

APPLESOFT SOURCE LISTING INFORMATION

As a supplement to the SOURCEROR program described earlier, a special program, SOURCEROR.FP is included on the MERLIN diskette. This utility will scan the resident copy of Applesoft present in your computer, and generate a fully labeled and commented source listing of Applesoft BASIC.

Because the completed listing is of a significant size, certain working files are DELETED during the compilation process. For this reason, you should NEVER produce your listing from the original MERLIN diskette. Instead, you should make a back-up copy of the diskette containing the SOURCEROR.FP program. Depending on the availability of double-sided diskettes, this may either be a separate diskette, or the reverse side of the original MERLIN diskette. Then use ONLY THE BACK-UP to generate the complete Applesoft source listing.

PLEASE NOTE, that this is NOT an "official" source listing from Apple Computer, Inc., but rather a product of the Author's own research and interpretation of the original Applesoft ROM. Apple Computer, Inc. was not in any way involved in the preparation of this data, nor was the final product reviewed for accuracy by that company. Use of the term APPLE should not be construed to represent any endorsement, official or otherwise, by Apple Computer, Inc.

Additionally, Southwestern Data Systems makes no warranties as to the accuracy or usability of this data. It is provided solely for the entertainment of users of the MERLIN assembler.

To use SOURCEROR.FP, place the diskette containing this file in the disk drive, and do a fresh boot in the usual manner. The HELLO program will then confirm that you are using a back-up diskette, and that you wish to generate the source listing. If you respond 'Y' to both questions, it will then proceed to generate MERLIN compatible files from the resident version of Applesoft present in either ROM or RAM card.

DO NOT INTERRUPT this process. It will take about 4 minutes to complete the process. When finished, the resultant files may be copied to any DOS 3.3 diskette.

GLOSSARY

ABORT	-terminate an operation prematurely
ACCESS	-locate or retrieve data
ADDRESS	-a memory location
ALGORITHM	-a method of solving a specific problem
ALLOCATE	-set aside or reserve space
ASCII	-industry standard system of 128 computer codes assigned to specifiied alphanumeric and special characters
BASE	-in number systems, the exponent at which the system repeats itself; the number of symbols required by that number system
BINARY	-the base 2 number system, composed solely of the numbers zero and one
BIT	-one unit of binary data, either a zero or a one
BRANCH	-resume execution at a new location
BUFFER	-large temporary data storage area
BYTE	-hex representation of 8 binary bits
CARRY	-flag in the 6502 status register
CHIP	-tiny piece of silicon or germanium containing many integrated circuits
CODE	-slang for data or machine language instructions

CTRL	-abbrev. control or control character
CURSOR	-character, usually a flashing inverse space, which marks the position of the next character to be typed
DATA	-facts or information used by or in a computer program
DECREMENT	-decrease value in constant steps
DEFAULT	-nominal value or condition assigned to a parameter if not specified by the user
DELIMIT	-separate, as with a : in a BASIC program line
DISPLACEMENT	-constant or variable used to calculate the distance between two memory locations
EQUATE	-establish a variable
EXPRESSION	-actual, implied or symbolic data
FETCH	-retrieve or get
FIELD	-portion of a data input reserved for a specific type of data
FLAG	-register or memory location used for preserving or establishing a status of a given operation of condition
HEX	-the hexadecimal (BASE 16) number system, composed of the numbers 0-9 and the letters A-F
HIGH ORDER	-the first, or most significant byte of a two byte hex address or value
HOOK	-vector address to an I/O routine or port
INCREMENT	-increase value in constant steps

INITIALIZE	-set all program parameters to zero, normal, or default condition
I/O	-input/output
INTERFACE	-method of interconnecting peripheral equipment
INVERT	-change to the opposite state
LABEL	-name applied to a variable or address, usually descriptive of its purpose
LOOKUP	-slang; see table
LOW ORDER	-the second, or least significant byte of a two byte hex address or value
LSB	-least significant (bit or byte) one with the least value
MACRO	-in assemblers, the capability to "call" a code segment by a symbolic name and place it in the object file
MICROPROCESSOR	-heart of a microcomputer, in the Apple, the 6502 chip
MOD	-algorithm returning the remainder of a division operation
MODE	-particular sub-type of operation
MODULE	-portion of a program devoted to a specific function
MNEMONIC	-symbolic abbreviation using characters helpful in recalling a function
MSB	-most significant (bit or byte) one with the greatest value
NULL	-without value

GLOSSARY

OBJECT CODE	-ready to run code produced by an assembler program
OFFSET	-value of a displacement
OPCODE	-instruction to be executed by the 6502
OPERAND	-data to be operated on by a 6502 instruction
PAGE	- a 256 byte area of memory named for the first byte of its hex address
PARAMETER	-constant or value required by a program or operation to function
PERIPHERAL	-external device
POINTER	-memory location containing an address to data elsewhere in memory
PORT	-physical interconnection point to peripheral equipment
PROMPT	-a character asking the user to input data
PSEUDO	-artificial, a substitute for
RAM	-random access memory
REGISTER	-single 6502 or memory location
RELATIVE	-branch made using an offset or displacement
ROM	-read only memory
SIGN BIT	-bit 7 of a byte; negative if value greater than $80
SOURCE CODE	-data entered into an assembler which will produce a machine language program when assembeled

STACK	-temporary storage area in RAM used by the 6502 and assembly language programs
STRING	-a group of ASCII characters usually enclosed by delimiters such as ' or "
SWEET 16	-program which simulates a 16 bit microprocessor
SYMBOL	-symbolic or mnemonic label
SYNTAX	-prescribed method of data entry
TABLE	-list of values, words, data referenced by a program
TOGGLE	-switch from one state to the other
VARIABLE	-alphanumeric expression which may assume or be assigned a number of values
VECTOR	-address to be referenced or branched to

SAMPLE PROGRAMS

The first group of three programs are superb Assembly Language utilities written by Steve Wozniak and Allen Baum, and are still found on the original Integer Basic F4 ROM. They are Supplied in source code format on this diskette for the benefit of Apple][Plus owners who do not have Integer Basic. They may be located at any convenient memory location.

The Floating Point Routines

These are single precision floating point routines that may be interfaced to a BASIC or assembly language program. Information on their use may be found in the source listings themselves.

The Multiply /Divide Routines

These routines are intended to be used as subroutines in assembly language programs, providing a four byte multiply or divide result. Brief information on their use is provided in the source listings, and a multiply demo by Dave Garson included on this diskette.

PRDEC

This is one the most used subroutines in the Integer Basic ROM set. It is called by virtually every routine which requires the output of an integer number in the range 0-65535. It is easily integrated in any Assembly Language program. To use it, load the accumulator with the high order byte of [number], load X with the low byte and call PRDEC. Alternatively, store the high byte in $F3, the low byte in $F2, and call PRDEC+4.

MSGOUT

This is a subroutine by Andy Hertzfeld to output ASCII strings from an Assembly Language program. If MERLIN INV or FLS Pseudo-ops are used in connection with it, the ORA #$80 must be removed, and all normal ASCII must have the high bit set. Also in the same source file are two simple subroutines to read ASCII and hexadecimal characters input by the user.

UPCON

This utility by Glen Bredon is provided for users who do not have a lowercase video display chip. It will search for source file comments beginning with either "*" or ";", and convert all lower case characters to upper case. Load the source file with MERLIN, then BRUN UPCON.

GAME PADDLE PRINTER DRIVER

When the Apple][was first developed, there were no printer interface cards, nor was there really much consideration even given to the need for a printer. Obviously, the folks at Apple computer had a requirement to hard copy their development routines, thus a primitive teletype driver was written by Randy Wiggington and Steve Wozniak to serve their in house needs. This was subsequently published in the famous "red book" instruction manual, the second for the Apple][. Along came the Disk][, and lo and behold, the driver would not work, since it ignored DOS and set its own I/O hooks. Next the Aldrich brothers took care of this problem, and we were back in business. By this time, of course, there was no desperate need for a game paddle driver; interface cards were developed, and worked well. Nevertheless, some users continued using the game I/O driver, so Dave Garson and Val Golding again modified the drive so that inverse and flashing characters would not upset the printer when doing a catalog, etc.

Concurrently, many new interface cards of all kinds were developed for the Apple, clock cards, 80 column cards, ROM cards, etc., until card space is now at a premium. Running a serial printer from the game I/O port is one such way in which the user can save both the cost of a printer interface and the slot space it would occupy. Already the teletype driver has been adapted to such printers as Integral Data, Base 2, Heath H-14, and others.

As a last step, Glen Bredon has added a number of improvements to the driver: it can print formatted BASIC listings to any column width, starting with 1, it can be output with or without video, and the video may be left on, even when printing beyond 40 columns, something most interface cards can not do. These functions are handled by BASIC POKE statements to the flags at the end of the program.

Full documentation and instructions are contained in the source file included on this diskette. Naturally, it is completely compatible with MERLIN, and called with the MERLIN USER command. This is set up when it is first BRUN, which establishes the ampersand hooks, which may also be used from BASIC.

In addition, the source code is well commented, so that it, in itself, serves as a tutorial on writing driver routines for different applications, etc.